51菜鸟到ARM高手进阶之旅

目　录

内 容 简 介

《51 菜鸟到 ARM（STM32）高手进阶之旅》内容非常丰富，以新颖的思路、深入浅出的带领读者从 51 基础入门到 51 精通一直达到 ARM 高手的水平。书中多处内容都是由作者 10 多年工作实践中总结而来。本书主要介绍 51 单片机和 ARM Cortex-M3 系列 STM32 的原理及应用， 全书共 6 个篇章。第 1 篇主要是教读者从零开始做一些 51 单片机的 DIY 项目；第 2 篇介绍 51 单片机的理论知识；第 3 篇 51 单片机全方位实战篇，手把手的操作，循序渐进的详细说明，从简单到难逐步深入来对 51 单片机进行全面的剖析；第 4 篇开始介绍 ARM 领域，拥有 51 的基础已经可以理解 51 单片机跨度到 ARM 领域需要些什么;第 5 篇 ARM 实战篇通过几个精彩实战章节来真正学懂 ARM。第 6 篇介绍了许多行业内幕以及经验之谈。

《51 菜鸟到 ARM（STM32）高手进阶之旅》条理清楚，深入浅出，图文并茂，学习脉络环环相扣紧凑，贴近读者，非常适合广大学生，电子爱好者，公司产品开发者的培训教材。本书由彭震编著。

前 言

1.兴趣很重要

一本入门的书应该怎么写？我为这个问题一直苦思许久。兴趣是导师，所以一本书要想提起读者的兴趣，它必须有趣。单片机的技术要有趣，入门的方法要有趣，忘记那些呆板的学术风格，删除那些深奥难懂的专业术语。

2.解决初学者的疑问

51 单片机是电子行业以及嵌入式行业最基础入门的一个敲门砖，古语说得好，凡事开头难，很多嵌入式的初学者和爱好者无不在入门这个阶段付出了很大的代价，有的效果还可以，有的花了时间精力效果还不理想，主要的困难到底有哪些呢？我主要归纳了一下几个方面，如下：

1. 小张：HI，我是初学者，我想知道嵌入式领域到底该怎么样进入最好？从哪着手最容易学？
2. 小李：我想我入门没问题，但我想知道怎么样学习上手最快，我想还知道怎么样学得更加深入，因为我想尽快成为高手
3. 小发：专家，我想问下学嵌入式是不是都要先学 51 单片机呢？
4. 小刘：据说现在 ARM 非常流行，我可以跳过 51 直接学 ARM 吗？那样不就省了学 51 的时间了嘛？嘿嘿
5. 小袁：我正在学 51 单片机，请问什么时候可以开始进入 ARM 的学习呢？我需要学到什么样子才可以进入到 ARM 的学习？
6. 小庞：我还是门外汉，嵌入式领域的那些牛人到底是怎样炼成的呀？这个领域到底有多深的水？
7. 小梁：这个领域的有哪些职位？这些职位各有什么责任和关系？我以后将怎么发展？

可以看到，这么多问题，每个人都会从自己的角度问一个不同的问题，而这些也确实存在的问题，不同阶段根据各自不同的情况，就会产生不同的问题，这些就是目前嵌入式领域资料过于杂乱，而缺少真正以人为本的指导而造成的，走了太多的弯路。

51 单片机是基础，麻雀虽小但五脏俱全，所以磨刀不误砍来功，大家不要认为学 51 是浪费时间，我个人觉得这恰恰是节约时间，为什么这么说呢？因为在日后的工作中，如果基础扎实的要比基础差的同学学习新东西更快，解决问题更快，接受和消化能力也就更强；比如张无忌有了九阳神功的底子，他在石洞里练乾坤大挪移就只花了 2 小时，而普通人可能要几十年，那个时候估计他早饿死在洞里了，所以基础很重要，你今天耐心的打下基础，就好像对土壤播下了种子，日后一定会有成功的果实收获。

另外一个方面，关于什么流行，什么不流行，我个人不是很赞同这个说法，实际上懂了硬件懂了嵌入式的人都知道，硬件懂 51，懂电路板设计，模电，数电，无非就是这些；软件领域操作系统，网络等，其他都是触类旁通的，有人说 ARM 最流行，最早我听说是 ARM9，ARM11，到后来是 A8，A9 等，到底什么最流行？什么最重要？其实你经历得多了，就知道还是基础最重要，新出的芯片无非是一个内核变了，这个可能是 ARM 公司的设计，但它的外围比如 I2C，SPI，串口等都至少是 20 多年没改过的，都是国际标准。唯一的差别就是体现在芯片的管脚脚位变了，速度变了，其他像通信协议这些一点都没变。

其他几个问题，关于牛人怎么炼成的，关于职业生涯的问题，书中有专门的章节或者穿插的内容在边说知识，边解释这些内容，帮初学者真正解决他心中的疑惑。

3.人性化的顺序

看看其他的入门书籍，闭上眼睛，你都可以猜到先介绍什么是单片机，然后介绍单

片机的历史，再后来介绍超级理论的内部硬件架构，再介绍非常理论的编程知识，最后找来十几二十个实验例程作为练习，学完之后感觉自己一直在看书，没有实际动手，学完感觉还是很迷茫。这样的教学顺序真的能事半功倍吗？对此我们在本书中是下了功夫研究的。看看本书的章节顺序你会发现与众不同之处，顺序的设计不是为了让目录看起来更工整，而是完全按照初学者的思维方式而编排。有一些动手制作和基本知识放在了本书的前面，慢慢深入，把一些更深的原理知识则放在了后面介绍。有些知识放在前面有助于后面内容的理解，有些知识放在后面可以让你有继续阅读的动力。试试我为你量身打造的新入门顺序，你会感觉到时刻充满的着激情和动力。

4. 图文并茂

书中很多图可以让初学者很快感受到身边存在一个实战的学习氛围，比如那些动手制作的实验，当你看到器件准备的图片，图片里由一些散件，慢慢组合成一个个实际的实验，使得不仅仅只是看文字去理性理解，还有一种感性的感知，这样的好处可以使得在实际动手时可以最大限度地减少想象力所带来的误差。

5.不只是技术，还有职业生涯规划

一般的书都是仅仅只偏重谈技术细节，而这本书不仅谈到软件技术和硬件技术，还带你从技术理念提升更高的层次；例如，软件高手是怎样的？硬件高手是怎样的？硬件设计的深度到底有多深？从技术高手慢慢过渡到高级管理层，各个角色在实际的公司中的作用，这些内容早一点知道，可以早点为自己做职业规划，也可以少走弯路；梨子到底是什么味道，你原本要尝 5 个才知道，看完本书后，在你今后的职业道路上，或许只需要尝 2 个或者 3 个就明白了。

6.解决了从 51 到ARM的过渡的难关，做了许多的衔接和铺垫

51 单片机的书一大堆，ARM 的书也是一大堆，先不管好与不好，但 51 与 ARM 衔接的书市场上非常少，在这个竞争激烈的时代，好的工具当然就是一把尚方宝剑。以前只有 51 所以很多人都会 51，后来出来了 ARM，这批会 51 的牛人很自然的就学会了 ARM，但是新出来的初学者呢？他们面对的眼花缭乱的 51 领域和 ARM 领域，资料目前如此之多，感觉学 51 不够强大；或者先学 51 又学 ARM 有感觉有种心浮气燥，巴不得马上懂得最高水平的 ARM 技术；只学 ARM 而不学 51 又感觉基础有点不踏实…..困难非常多，想法非常多，拿到这本书就像获得了一盏明灯，它可以协助你走完这最艰难，最无助的路程，这段路程中的你可能提出个问题，都会让人笑掉大牙；你走完这段路之后，就已经成为了半个专家，你

已经有了自己捕食的能力，你已经找到自己的发展方向。

7.完整的配套和升级服务

目前正在努力建立本书完整的配套服务，包含提供问题解答，资料更新，例程增加，视频教程，硬件模块增加；一方面增加更多的 DIY 教程，提高大家的动手能力；另外一方面增加一些实际项目教程作为辅助的资料丰富大家的经验和独立操作能力；因为书本篇幅有限，所有这些资料将会通过博客网盘公开免费下载；另外，每年会在假期组织一到两次封闭式免费培训活动，作者将面对面的讲解本书知识，听取大家的反馈意见。

8.致谢

感谢我的导师—国防科学技术大学计算机学院姜新文教授对本书整体方向上的指导；感谢同事贾力瑛，杨锐，黄毅，罗剑伟，谢光义，杨大云，肖敏锋，庄伟，李宝勋，李壮永，方辉，吴秋平，常洪雨，蒋德为，胡明城，李学奎，罗召杰，刘文发，胡超对本书编写过程中提供了非常大的技术支持和内容支持；感谢国防科学技术大学的同学陈树根，李海军，许良栋，黄宗成，陈胜良，朱贤良，黄波，叶湘斌，熊橙梁，易鸿斌，黄栋，袁海波，覃星，向晟，张晟，郭亮，蔡卉佳，刘欢，范可在本书编写过程中所提供的来自腾讯、百度、阿里、华为等各优秀公司的学习方法和一线工作经验；感谢我的好朋友张韬，薛君茹，蔡振闹，陈刚，钟立雄，邓志伟，钟锦辉，王丽丽，古政在本书创作期间给予从电子行业角度和嵌入式培训行业角度所提出的深刻建议和具体帮助；尤其要感谢北京航空航天大学出版社胡晓柏主任和编辑工作者的大力支持，在此表示衷心的感谢。最后要感谢我的父母和所有支持嵌入式和电子行业发展的兄弟姐妹朋友们。

由于时间仓促，加之水平有限，书中难免出现错误和缺陷，敬请各位读者批评指正。如果读者有兴趣，亦可加入 armjishu.com 论坛或关注 http://blog.sina.com.cn/stm321 与作者交流（E-mail:armjishu.com@163.com）。

彭 震

2017 年 5 月

整本书阅读的指导必读

本书内容比较丰富，侧重于实践，如果使用适当，将是目前国内唯一一本既包含基础又有深度的不可多得的奇书。这本书将一个从未入门的嵌入式学员，在短时间内突飞猛进，作者历史大约 10 年时间打造成这本黄金手册，该手册还有专门搭配的光盘资料辅佐，许多篇幅的内容无法一一在书中展现，只能保存在光盘中，本书贯穿整条学习路线，跟着书本一起走，将可以贯穿整本书的精华。

网络上各种资料看得眼花缭乱，对于目前海量的互联网，实际上资料多了，反而浪费时间；本书的目标是让初学者真正学会，让会的人学懂，从 51 单片机步入到 ARM 的高端境地，让不会的人迅速进入状态，让懂的人拓展自己的知识面，本书总共分六大篇幅：

- 《第一篇 从零开始动手制作 51 单片机小项目篇》
  主要内容：如何从 0 开始手把手的搭建一个一个实验，很多书都是从理论开始，这样的缺点是模模糊糊，我们这里是从动手制作开始，通过实践去感受单片机的魅力，然后再学理论，这样的效果是事半功倍的，在很多初学者爱好者中屡试不爽。
  适用群体：0 基础的初学者
- 《第二篇 51 单片机理论深入篇》
  主要内容：有了第一篇动手的基础之后，再适当的补充一些理论方面的知识，这些内容都是比较深入浅出的描述了单片机的真正本质原理，而几乎不参杂那些传统书籍的生搬硬套的理论，这是真正活的灵魂，建议一定要看看。
  适用群体：初学者或者有基础的人都建议阅读
- 《第三篇 51 单片机全方位实战篇》
  主要内容：全部都是手把手的操作以及由简单到难的程序，逐步深入，无须多想，按照书籍的步骤来，一天阅读一部分，动手操作一下，学习轻松，覆盖全面。
  适用群体：初学者或者有基础的人都建议阅读
- 《第四篇 ARM 理论基础深入篇》
  主要内容：经过一段时间的苦练，终于到了 ARM 理论基础深入篇，首先要恭喜自己的努力成果；对有 51 基础的人来说，ARM 是一层神秘面纱，ARM 意味着更高端，意味着由 51 的小项目进入 ARM 的大项目，这里的理论将从一个专家的角度去帮你分析全方位了解到 ARM，从理论的角度了解 ARM 的面貌。
  适用群体：有 51 基础的人
- 《第五篇 ARM 实战篇》
  主要内容：开始实际来操作 ARM，详细的剖析，如果您掌握了，那么您真的就从 51 入门敲开了 ARM 的大门，成为了一名高手。
  适用群体：有 51 基础的人

- 《第六篇 嵌入式高手进阶之路》
  主要内容：侧重于理论，您站在巨人的肩膀上看看外面的世界，看看如何成为高手或者成为高手路上有什么障碍，这样在以后的超出本书范围内去学习，可能会为您指明一条道路，俗话说，少走弯路也是一种进步。
  适用群体：各界人士

- 《神舟 51 单片机的全套视频教程》已经推出
  在学习 51 单片机的时候，与文档一对一的视频教程一起学习效果更好。

第一篇 从零开始动手制作 51 单片机小项目篇

1.1 我的单片机自学经历

单片机是什么？刚开始我并不知道，记得小时候有一次在电子游戏室玩一个叫拳皇的游戏，玩到一半时游戏机坏了，维修师傅跑过来把游戏机打开维修，此时可以一块绿色的电路板

上有着许多黑色的芯片，还有各种电子元器件，那时候我很好奇的看着这个电路板，想着这么好玩的游戏就装在这个电路板里，实在是太神奇了。

后来在慢慢长大，我开始知道了，原来那些电路，还有那些黑色的芯片都是一种叫单片机的芯片来控制的，单片机，单片机，单片机……这 3 个字已经重复一次又一次的在我的脑海里回荡起来。

终于我读大学了，走入大学的图书馆，看着整齐的书架上摆放着密密麻麻的书籍，这里藏书到底多少册我真不清楚，反正整栋楼都是书，《单片机接口技术》、《单片机原理及其接口技术》、《51 单片机应用开发范例大全》，慢慢 4，5 个书架的书，这真是嵌入式的天堂，立马感觉到一种知识的海洋，感觉到对面的海风吹来书本的味道，心灵感觉特别的宁静。随手抽了一本单片机基础教程看了看，除了前言能看明白之外，其他的那些图包括代码分析都基本是天书，而且越看感觉越难，好深奥的知识。

随着我怀着非常沮丧又激动的心情，去翻阅着其他的书籍，真不相信找不到一本合适的，让我入门的，找啊找啊找，找了完了一个书架，继续找，找遍了，3 个小时过去了，我的腿和胳膊都很酸，还是一无所获，除了知道了许多名词和专业术语之外，没有任何的收货；旁边的书架就是 ARM 的，天啊！我感觉自己已经被各种知识的海洋包围着，找不到突破口。

不管那么多，先抱着几本书，借回去看看；然后开始咨询我的师兄，参加了一个培训班，突然发现培训班讲课的顺序怎么与书本的内容不完全相同，培训班更加浅显易懂得多，听完培训班的课程再回家仔细阅读书籍，这样才读懂了第一本书，算是入门了，所以我想说的是，现在书籍确实非常的多，但好的书，适合自己的书真的是太少太少了。

我估测了一下目前市场上的书，大约有 70%以上都是讲原理，比较难入手，这么说吧，这些书如果懂的人一看就懂，不懂的人看了也还是不懂；还有 20%的是讲实例，这样的书切入点是比较好的，刚开始我很激动，后来仔细研究深入到内容里面的时候，发现这些实例都非常的冗长，要么就是注视不全，要么就并不是专为初学者定义的，比较难看懂，看懂了缺少细节的点评，也很难学懂东西，俗话说，还没学会走，就让你学怎么跑，虽然能跑了很激动，但是基础一点都不扎实；最后剩下 10%的好书，可以入门，其实说实话，学东西离不开实践，有实践才能学到东西，这个是硬道理。

后来，我发现，从零基础到入门这一个霎那间反而是最难的，那些复杂高深的项目反而不是最难的，因为那个时候你已经有基础了，足够基础自己研究去学习了；而且那个时候兴趣已经变成了主导地位，只有从零基础到入门这个阶段，是痛苦的，是枯燥的，有种想跳楼的感觉，但是每次我都是用喝酒替代了！哈哈，不要害怕，如果你真想学好单片机，只需要做好心理准备，100 米短跑比赛前 10 米是最难的，后面 90 米都是容易和简单的，之前的 10 米是最最最重要的，也是最最最艰难的，做好心理准备，每进步一点就给自己一些鼓励，再准备一些酒，就可以学好学精通单片机了。

现在非常幸运的是，我们把这个经验整理并完整的设计成了神舟 51+ARM 开发板的书籍，通过理论加实践双重学习来尽快掌握单片机，原本要花费 1 年-2 年时间才能入门到精通的，现在只需要花费 3 个月时间（这里需要提到的，7 天学会单片机那种概念不知道是谁提出来的，一点都不切实际，你可以看看我们的这本书，书中句句都是干货，看完都不止 7 天了，而且这本书里还有许多内容，更深入的内容都是点到为止，我们只能领大家进入到一定的程度和水平，剩下的就要靠大家去发挥和深造了）

1.2 我眼中的单片机

单片机就是一块黑色芯片，一个智能电脑，它懂得的一切都是由我们来吩咐的，它就像是一个听话的小孩子，我们只要输入一个叫它干什么的程序，它就可以按照你的吩咐为你服

务了；你要它做另外的事情，就输入另外的程序就可以改变了。

那它到底可以干什么呢？难道可以帮你洗衣做饭吗？当然，其实我们现在生活中的电器大多都用到了单片机。我们的洗衣机里就用到了单片机控制，可以设定好洗衣时间和方式，按下洗衣键，它就会按照你的设置按时注水，洗衣服，甩干脱水，洗完最后还滴滴几声或者唱首歌曲告诉你衣服洗完了；我们家中的自动电饭煲也用到了单片机，由它控制加热、时间，煮出香香的米饭，有的电饭煲还可以用来煲汤，煲粥，都是自动设定就可以啦，哈哈！这样一来单片机真的可以实现为我们洗衣做饭了。

单片机是用程序进行控制的，需要做什么用途，就做什么程序进去就可以了，再配上各种不同的外围电路，控制各种不同的电器设备；要跟电路打交道，如果大家对电路制作有一定基础的话，学好单片机就变得更加的容易，如果没有这个基础，可以跟着我们一步一步来也能达到学会单片机的目的。下面列出一些单片机的实物图如下图 1-1 所示：

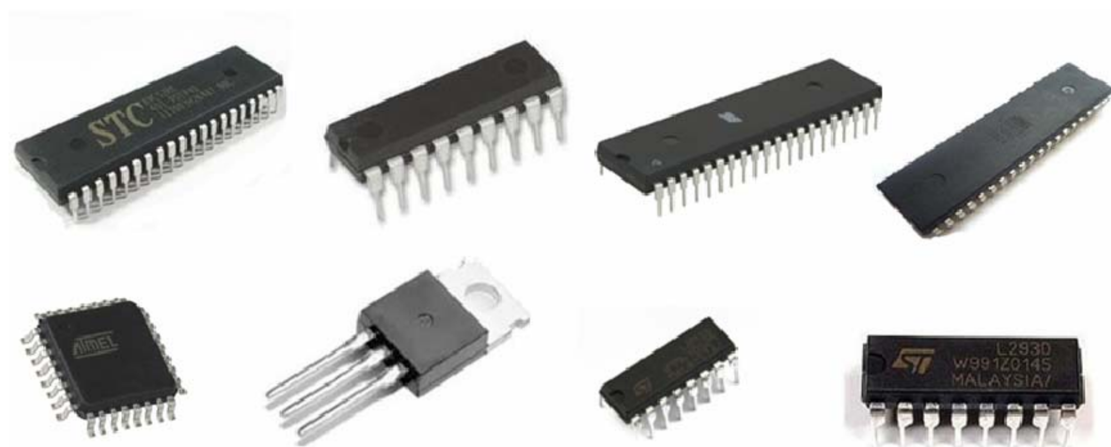


图 1-1 单片机芯片实物图照片

图 1-2 中列出了一些学习使用 51 单片机制作的各种小产品，有小车，电子秤，简单的逻辑分析仪等：



图 1-2 学生电子大赛作品图

以上都是一些简单产品制作，单片机还有更多的广泛用途，比如神舟十号飞船上很多电路控制都是由单片机实现的，还有 GPS 导航仪，门禁系统，智能家居控制，LED 广告灯等，都是由单片机实现的。

1.3 零基础搭建一个最小电路

1.3.1 搭建电路的地盘

什么是地盘？有点黑社会的感觉，确实，万丈高楼平地而起，建一栋楼也需要一块地；搭建一套电路也需要一个电路板，这个电路板上放各种各样的元器件，然后相互连在一起。在电路板中，有万能板和面包板两种，万能板就是许多的洞洞，这些洞洞之间都是单独的；而面包板呢也是有非常多洞洞的板子，面包板与万能板不同之处主要是面包板虽然正面布满孔洞，但它们中的每一个孔洞都不是独立的，而是按一定的规则连接在一起的。可以参看下面的面包板的实物图 1-3 照片。



图 1-3 面包板实物图照片

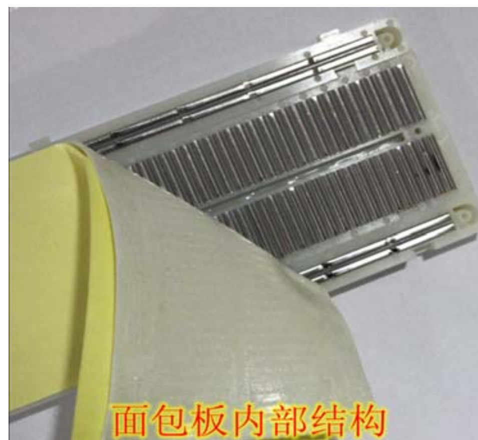


图 1-4 面包板内部实物图照片

我们用手撕开面包板的表皮之后，可以看到每行的过孔都是连接起来了的，面包板内部实物图照片如下图 1-4。

市场上常见的面包板是对称的双排结构，它们的行之间是连接在一起的。两侧的电源连接口是列向全部连接在一起的.但还有一些面包板的电源连接口是列向分几段连接在一起的，购买的时候要问问店老板或者回家使用万用表测量。可以看到下图 1-5 用红圈标记好了面包板内部电路的连接关系。

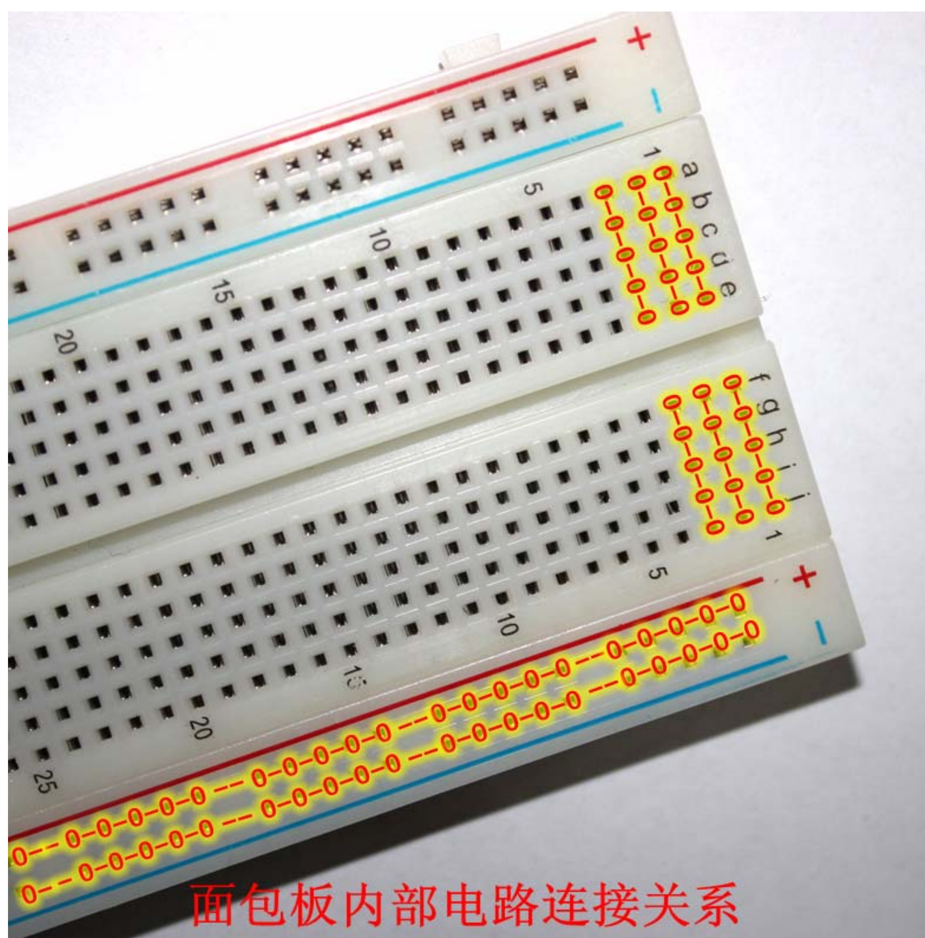


图 1-5 面包板内部电路连接关系

1.3.2 单片机要运行的最简单条件

一个单片机要运行，最基本的条件是什么呢？大家想想，首先一定需要电源吧，没有电单片机怎么能运行呢，所以肯定是需要电源的，这里的 51 单片机 5V 供电的，所以外围要加一个 5V 的供电电压。

其次，单片机运行是有一定速度的，比如 1 秒钟运行多少次就叫做 1 秒钟执行多少次频率，既然涉及到的稳定的频率，那么就会有一个时间周期，多长时间运行一次，如果不规定这个时间周期，那单片机就不稳定，无法有节奏稳定的去执行一个任务，所以这里肯定是需要一个时钟；而这个时钟一般在外围，时钟目前是由一个晶振来提供的，但是有的单片机内部也有时钟，外部的时钟如果坏了，或者不接外部时钟的时候，内部时钟也能够运行（有的单片机自己内部就有一个时钟，有的没有），比如 STC 公司出品的 STC12C2052 单片机就是具有内部时钟的单片机。

有了电源，有了时钟，还需要什么呢？当单片机启动的时候，或者在运行过程中出现不稳定的情况怎么办，是不是重新开始运行呢？对，重新开始运行，那就需要重新上电，但当单片机通电上电的时候，内部有些部件（单片机内部是由许多各种部件组合而成的）因为达到正常工作电压的值需要有个很小的延时，所以刚一重新上电有些部件可能工作还没有到正常值，必须要等到电源的电压稳定之后才可以，所以这里需要不断的进行复位操作，直到电

压稳定后，复位就停止，单片机才能正常工作，所以不能缺的还有复位电路，比如 STC 公司出品的 STC12C2052 单片机内部集成了复位电路，这样就不需要外接复位电路了，大家到这里一定还是不太明白，毕竟是理论上的，不过不用担心，接下来我们会有一些实际的操作来让大家更加熟悉。

1.3.3 动手搭建最简单电路点亮一个LED灯

通过一系列选型，我们打算用 STC 公司出品的 STC12C2052 这颗单片机 CPU 来举例，这颗芯片内部集成了时钟电路和复位电路，就只差电源了，我们只需要外部供给它 一个电源那单片机就可以正常工作了。

下面我们将用极少元器件组建一个最精简化电路系统，让初学者看到单片机点亮一闪 LED 实验。最精简化的电路系统是什么样的？下图 1-6 是元器件清单：

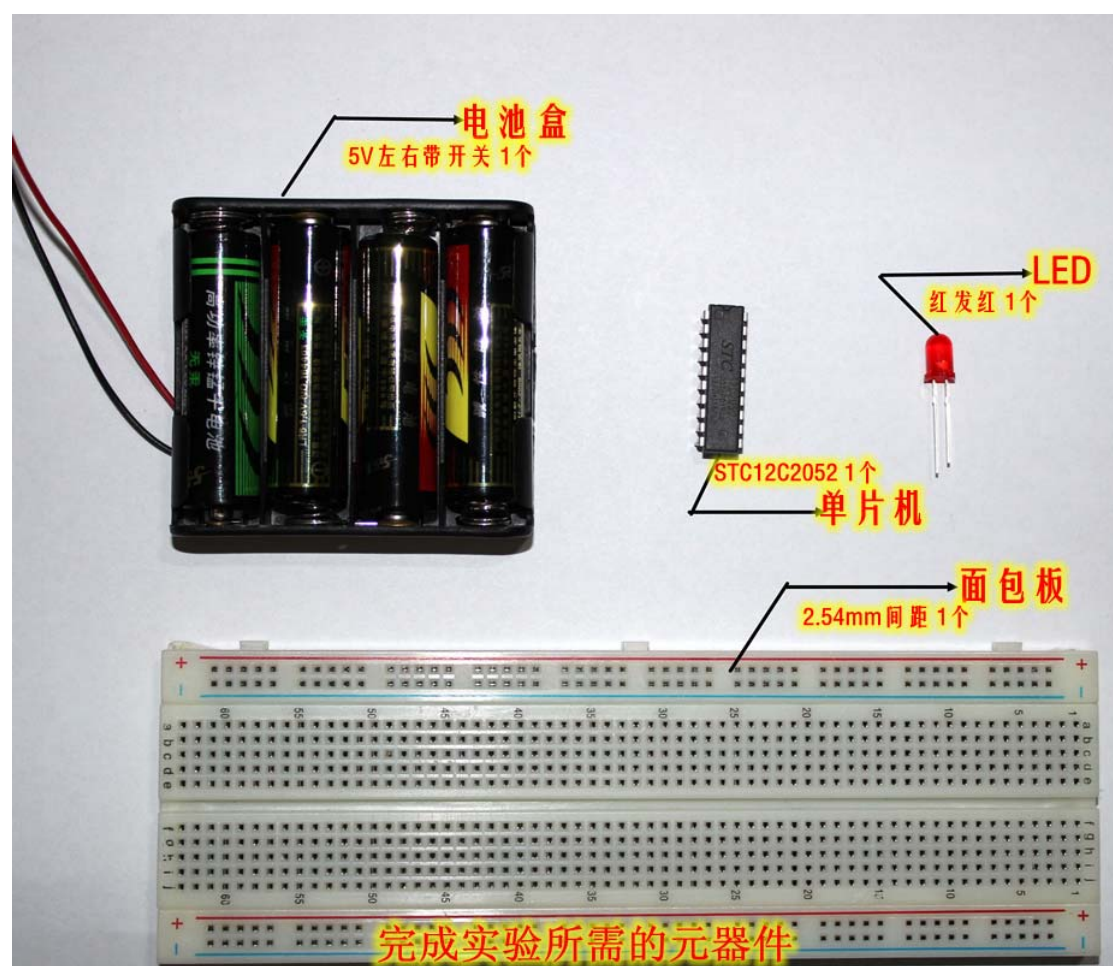


图 1-6 LED 点灯实验元器件清单

STC12C2052 （在选材上我们选用 STC12C2052 单片机它内部集成复位电路、晶振时钟电路），3 节 7 号电池（选用 3 节普通碱性电池来提供电源，减少了利用市电供电的降压、稳压问题，因为单片机的供电需求是 4.5~5V 左右，3 节干电池大约是 4.5V 所以是刚好可以满足要求），1 只 LED 灯（发光二极管），1 块面包板---2.54mm 间距；再来看最小系统工作必备要求看看我们是否达到：

1）**电源**---我们选用 3 节普通碱性电池来提供电源 4.5V，可以满足。

2）**晶振**---我们选用 STC12C2052 单片机它内部集成晶振时钟电路。

3）**复位电路**---我们选用 STC12C2052 单片机它内部集成复位电路。

1 块单片机、1 个电池盒、1 只灯、1 块面包板。实验变得如此简单，就连数字电路入门也不能与之媲美。正因为元器件极少.所以制作简单、快速，可以在 10 分钟之内完成制作并看到实验效果.顺便说一下，学习单片机是必须要有 1 台电脑的，这一点没有任何商量的余地。你可以是已经有了电脑之后来学习单片机的，也可以为了学习单片机购买 1 台电脑。

下表 1-1 列出了这个实验所需要的所有器材，包括名字，型号，数量，价格等，可以凭这个列表去购买到所有的器件，就可以做好这个实验了。

表 1-1 LED 灯点亮实验器件列表

品名	型号	教量(个)	参考价格（元）	备注
电池盒	3 节 7 号	3	4.50	可选择其他电池，保证输出电压在 4.5∽5v
单片机	STC1202052	1	6.00	可用 STC12C2052AD 替换
LED	直插 5mm	1	0.20	可选择各种其他频色和型号的 LED
面包扳	2.54mm 间距	1	12.00	可以在上面灵活搭建自己的电路

STC12C2052 单片机芯片管脚图和芯片实物图如下图 1-7 所示：

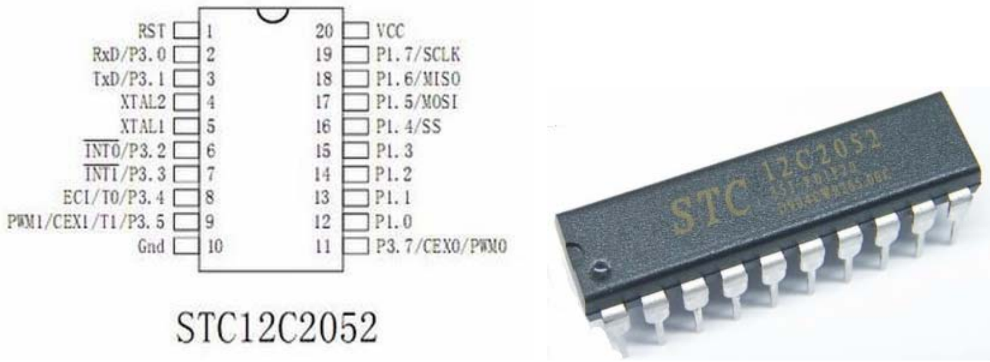


图 1-7 STC12C2052 引脚定义和实物图

我们使用的单片机 STC12C2052，它的工作电压是 3.5V∽5.5V 分工业级（I）和商业级（C）的产品；注意，工业级和商业级的主要区别是温度差值不同，工业级适用的温度更宽范围，当然其他的一些芯片在其他方面也有区别，但大部分的芯片主要是温度的差异。

我们仅仅只是实验，用哪一种都可以。从引脚定义图来者，最小系统需要最关心的电源是第 20 脚是电源正极（VCC)和第 10 脚是电源地（GND)两个，VCC 和 GND 是天生一对，要接就得都接上。

可以看到管脚有 P1.0~P1.7 以及 P3.0~P3.5,例如第 15 脚是 P1.3,它是是单片机的一个 I/O 接口顾名思义就是 IN/OUT。写成中文就是输入/输出接口,这是单片机最基本的接口，可以说是单片机就有 I/O 接口。那输入、输出的是什么东西呢？不是别的.正是电平。电平是一个相对的概念。

关于电平，简单的说 1 个电路里有 1 个公共地端（GND），如果还有 1 个 5V 的电源(VCC)则 5V 是高电平，公共地端是低电平。如果还有 1 个-5V，那么-5V 和的两者比就是低电平。电平和身高一样，你自己一个人没有高矮的概念，你要是和 NBA 的迈克乔丹或者科比比身高

你就是低电平，他是高电平；你要是和 1 岁的小朋友比，你就是高电平，他是低电平。

1 个单片机芯片有电源和地（如果用 3 节电池供电就是 4.5V,但通常习惯上是用 5V 电源供电,用电池只是为了精简电路让初学者更容易接受，4.5V 也可以使得单片机正常工作了)，所以说 5V 是高电平，公共地 GND 是低电平。

“I/O 接口即可以输入又可以输出是什么意思呢？”。输入的意思就是外部电路通过 IO 管脚输入给单片机，让它知道我们输入的是高电平还是低电平，单片机会定时去采样这个管脚上的电平，知道是高电平还是低电平，这样我们就可以控制它了。例如可以做做这样一个程序，让单片机在检测这个管脚是高电平的时候把灯打开，证明我们回房间要看书了；在检测低电平的时候，把灯关闭，灯不亮了，证明我看完书要出去了。这样就非常的灵活，我们可以随心所欲的去根据外界状态的变化而变化。

输出也是一样，单片机可以自己输出高电平或是低电平。我们就可以写一个程序, 让它在 I/O 接口上输出高、低电平去控制一些东西，或者我们读出它的高、低电平状态来现察它在干什么。

一个单片机上有好多个 I/O 接口，我们现在用的这款 STC1202052 上就有管脚有 P1.0~P1.7 以及 P3.0~P3.5，还有一个 P3.7 总共 15 个 I/O 接口，当然还更多 I/O 接口的芯片，这里我们以这颗为标准，这颗芯片弄懂了，其他都是类似的。比如，可以写一个程序，让单片机专有几个管脚做输入，我们可以通过这几个管脚采集到外面传输进来的数据；另外还可以控制单片机的另外的几个管脚做输出，去控制另外要控制的东西，比如控制红外关闭空调或者打开空调吹吹。

举个小例子，比如我们在 1 个 I/O 接口上连接 1 个电灯开关，就假设这个 I/O 接口是 P1.3 吧 (第 15 脚)。开关的另一端接到 5V 电源（VCC)上。在另一个 I/O 接口上接 1 个小灯泡，假设是 P1.4 吧（第 16 脚）。小灯泡另一端接在公共地端（GND）。写一个程序告诉单片机，当我们接通开关时（P1.3 与 VCC 短接）则接在 P1.4 上的小灯泡点 亮（P1.4 输出了高电平），程序运行时，单片机就会不断地查检 P1.4 接口的电平状态，当 P1.3 接口输入为亮电平时(开关接通),单片机就会以迅雷不及掩耳之速度输出高电平给 P1.4 接口.让小灯点亮。这就是单片机丨/O 接口的功能之所在。，红色的线表示接电池的正极，黑色表示接负极，电池盒如下图 1-8 所示。



图 1-8 7 号电池元件盒

这个电池盒是容纳 3 节 7 号电池的，不仅体积小巧而且自带开关，在电子市场花费了 3 元买的，可以计算一下，1 节 7 号电池是 1.5V，3 节是 4.5V，单片机电路的供电电压是 3.5V~5.5V 的直流电源，所以刚好是够的；当然你也可以用其他的供电方式，比如 5V 的电源变压器，电脑的 USB 接口（USB 接口也是标准 5V 的电压）或是其他电源都是可以的。

如果没有买到电池盒，也可以自己拿 3 节电池连起来，用透明胶纸粘连起来，可以自己制作一个电池串，把正极和负极用导线连出来就行，也是一样的，不过最好是有电池盒，最好是有开关，这样就会比较方便，注意红线是正极，黑线是负极，不确定的话就用万用表测量一下或者用一个纽扣电池把二极管的两个管脚捏住，如果亮的话，那么纽扣电池的正极所接的二极管那端就是正极，另一端就是负极；另外看两个管脚的长短，长的那个管脚是正极，短的是负极。LED 长脚为正极+、短脚为负极，颜色搭配可以随意，用自己喜欢的颜色；下图 1-9 是各色二极管的样子，可以看到每个二极管的管脚并不是等长的。



图 1-9 各色发光二极管

开始动手了，首先把单片机轻轻固定到面包板的中央，固定位置请见下图 1-10. 接下来用一个装 3 节电池的电池盒，来给单片机芯片进行供电，单片机第 20 脚接电源正极（红线）第 10 脚接电源地负极（黑线），如下图 1-11 所示：

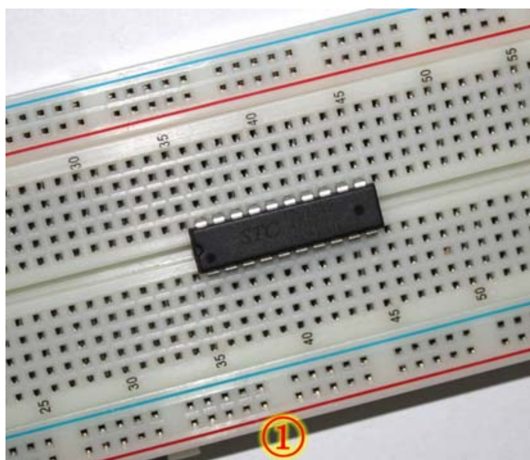


图 1-10 51 单片机芯片插在面包板上

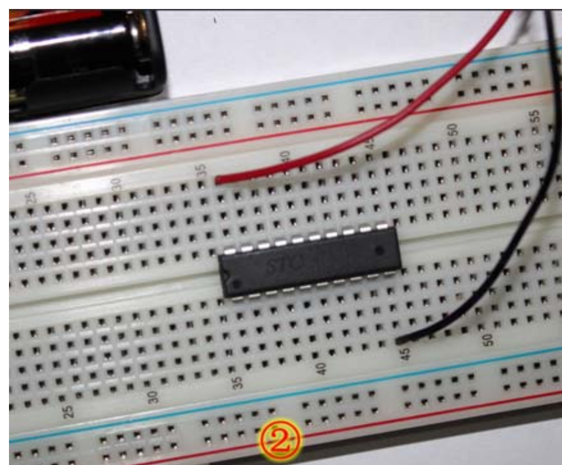


图 1-11 电池盒给 51 单片机芯片供电

接下来，把 LED 灯的正极插入到面包板中，注意 LED 灯的正极要与电池盒的正极一致，另外一只脚与 51 单片机芯片的另外一只脚相连即可，下图 1-12 为连接图：

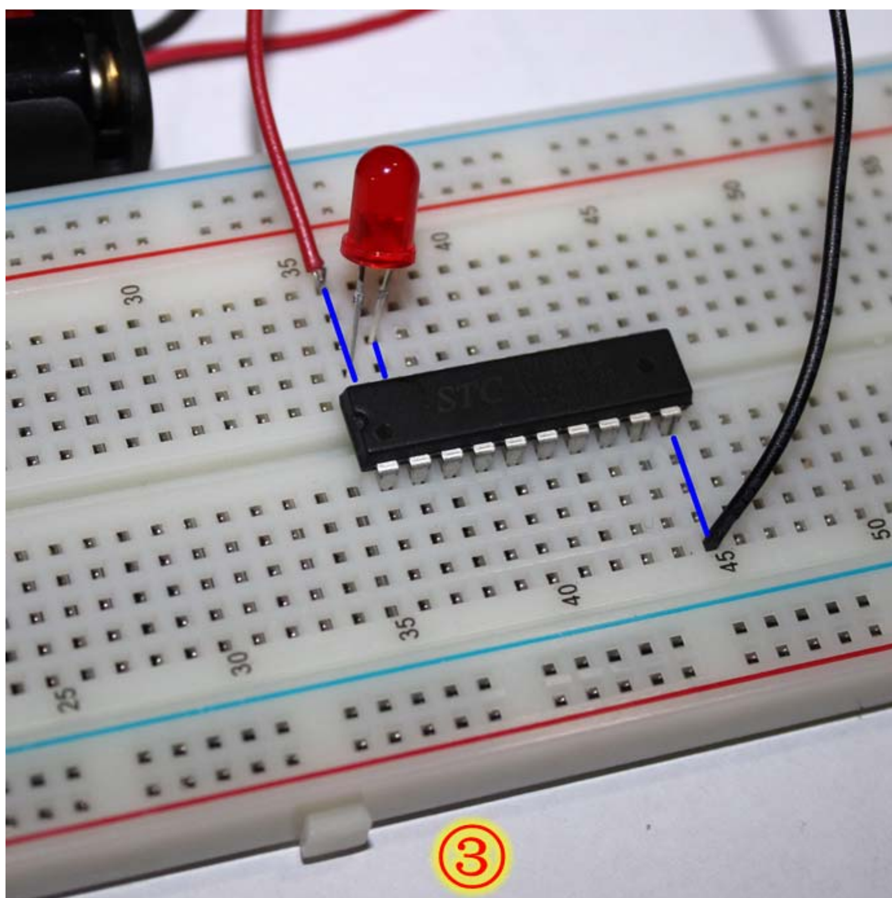


图 1-12 LED 灯与单片机电池盒连接上

LED 正极与单片机第 20 脚连接，负极与单片机第 19 脚连接（所有连接完成），下表是电池盒与 LED 灯和 51 单片机管脚的对应列表 1-2。

表 1-2 LED 灯点亮实验连接对应表

	极性	单片机引脚
LED 灯	正极（+）	第 20 脚
	负极（-）	第 19 脚
电池盒	正极（+）	第 20 脚
	负极（-）	第 10 脚

打开电源开关（OFF->ON）后，可以看到图 1-13 中 LED 灯非常明亮的进行闪烁，说明我们的实验成功了。这是因为单片机在厂商在生产的时候就写入了 1 个 LED 彩灯闪烁的小程序，就是为了快速验证单片机的好坏，也正好帮助我们完成了第 1 个单片机的实验，点亮效果如图 1-13：

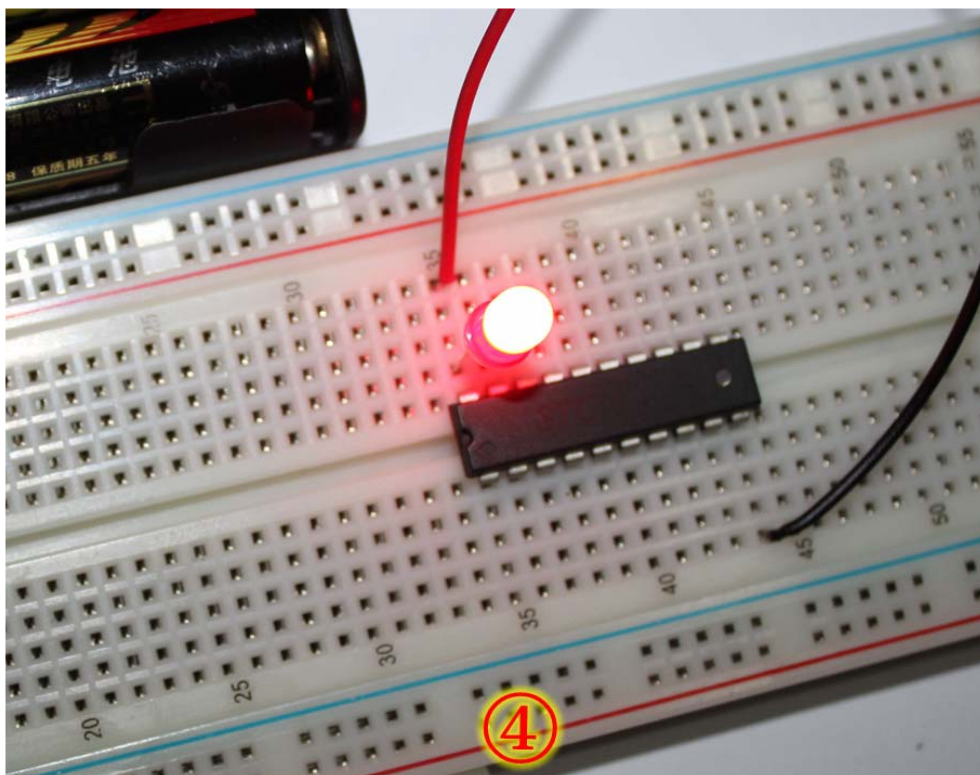


图 1-13 51 单片机点亮了 LED 灯

有人会比较贪心了，嘿嘿，就 1 个小灯一闪一闪亮晶晶有什么好玩的。别着急哈，下面我们来玩 3 个灯的。

1.3.4 点亮多个LED灯

单片机点亮一个 LED 灯是最基础最简单的对单片机管脚的操作；能够点亮一个 LED,还可以进行变通一下点亮一个闪烁的 LED，还可以点亮多个 LED 灯。

下面连接 3 个产生流水灯，有没有发现这次的亮度没有上一个实验中单个 LED 的高了。什么原因呢？这是因为之前那个实验是电池作为正极对 LED 灯进行供电，而这个实验是单片机的管脚自己做为正极为 3 个 LED 灯供电，而单片机的电是电池提供的，所以这个里面会有一些损耗存在，因为单片机的管脚输出的电压和电流跟电池的正极相比，还是有差别的，下图 1-14 展现了具体的连接情况：

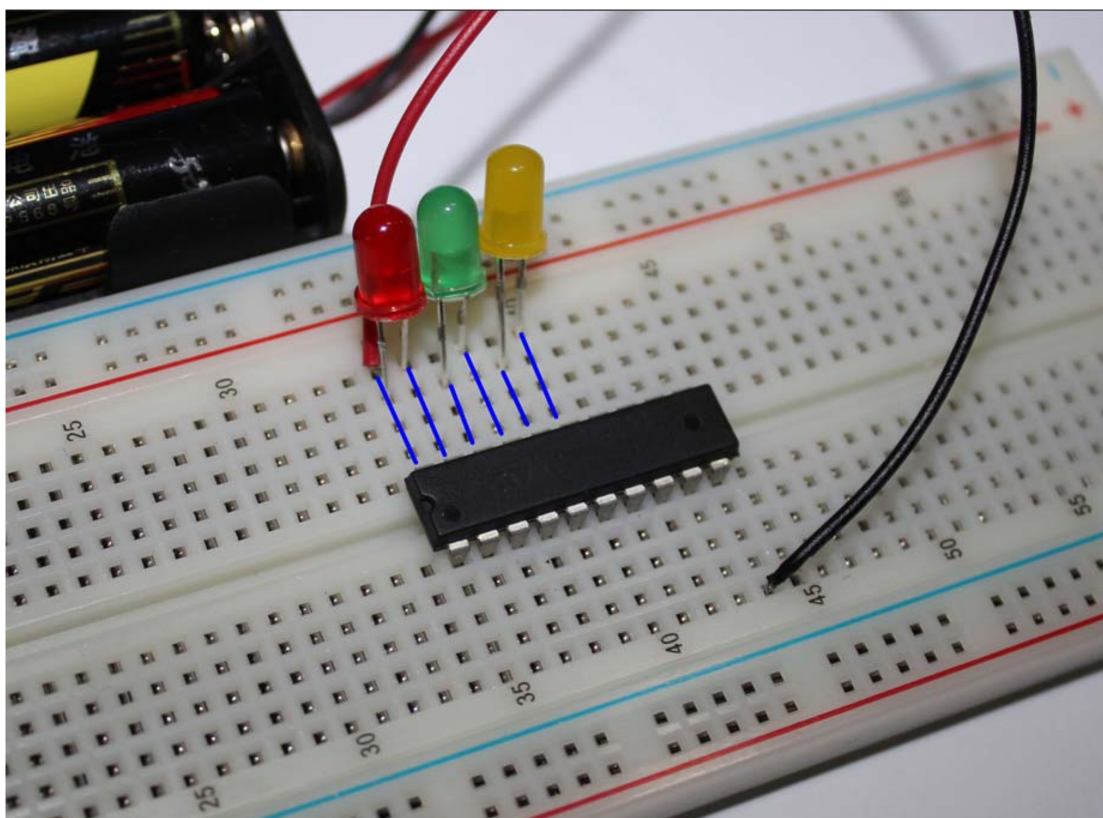


图 1-14 同时点亮三个 LED 灯

具体管脚连接对应可见下表 1-3：

表 1-3 同时点亮三个 LED 灯连接说明

神舟 51+ARM 单片机开发板 （开源实验）		
	LED 极性	单片机引脚
第 1 个 LED	正极（+）	19 脚
	负极（-）	18 脚
第 2 个 LED	正极（+）	17 脚
	负极（-）	16 脚
第 3 个 LED	正极（+）	15 脚
	负极（-）	14 脚
电池盒	正极（+）	第 20 脚
	负极（-）	第 10 脚

1.4 单片机怎么样下载程序

1.4.1 了解串口

前面几个点亮 LED 的程序已经看到了，但是并没有说程序是怎么样下载进去的。那么究竟是怎么样把程序从我们的电脑上下载到单片机里去的呢？对，通过串口，51 单片机可以通过一个叫串口的下载接口下载程序到单片机内部，那究竟什么是串口呢？我们见过串口这个神秘的东东吗？

这个要追溯到电脑出现时的潮流，其实传统的电脑主机箱后面就有串口接口，但现在的大部分笔记本电脑都没有，一般都只有 USB 接口了，串口好像在笔记本电脑上快要绝迹了，下面这个图 1-15 是最常用的 9 针串口座的样子：

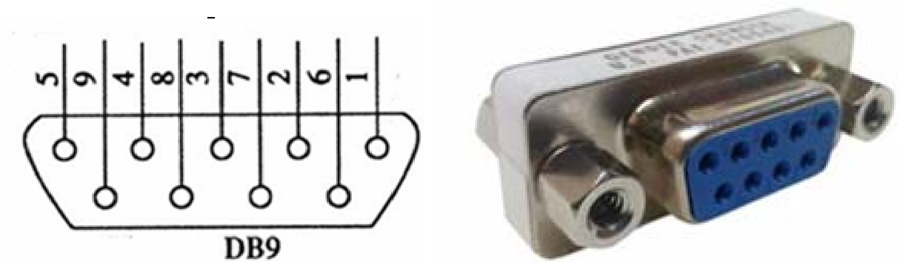


图 1-15 DB9 串口接口信号图

引脚功能说明如下表 1-4 所示。

表 1-4 DB9 串口接口引脚功能说明

9 针串口		
针号	功能说明	缩写
1	数据载波检测	DCD
2	接收数据	RXD
3	发送数据	TXD
4	数据终端准备	DTR
5	信号地	GND
6	数据设备准备好	DSR
7	请求发送	RTS
8	清楚发送	CTS
9	振铃指示	DELL

对于 51 单片机来说，我们在电脑上运行一个上位机软件程序，这个上位机软件程序可以把电脑上的单片机程序通过这个电脑主机的串口接口下载到单片机里面，当然前提是电脑的串口必须与 51 单片机的串口接口连上才行。

1.4.2 自己制作串口下载线

电脑的串口是标准 RS232 电平，RS232 电平的范围是+12V 到 -12V 的，而 51 单片机的电压是 0~5V，所以它们之间如果要通信的话，就需要做一个转换，比如把 RS232 电平转换成 51 单片机的可以接受的电平，这样就用到了一个叫做 MAX232 的芯片，这颗芯片在芯片内部就完成了这样的一个电平转换的工作。

如果你的电脑主机上有 9 针的串口，就可以直接跟着我完成下面的制作。没有串口的朋友可以考虑购买一根 USB 转 RS-232 的转换线，同样可以达到目的。下面我们自己制作串口下载线，首先准备如下图 1-16 这些配件：

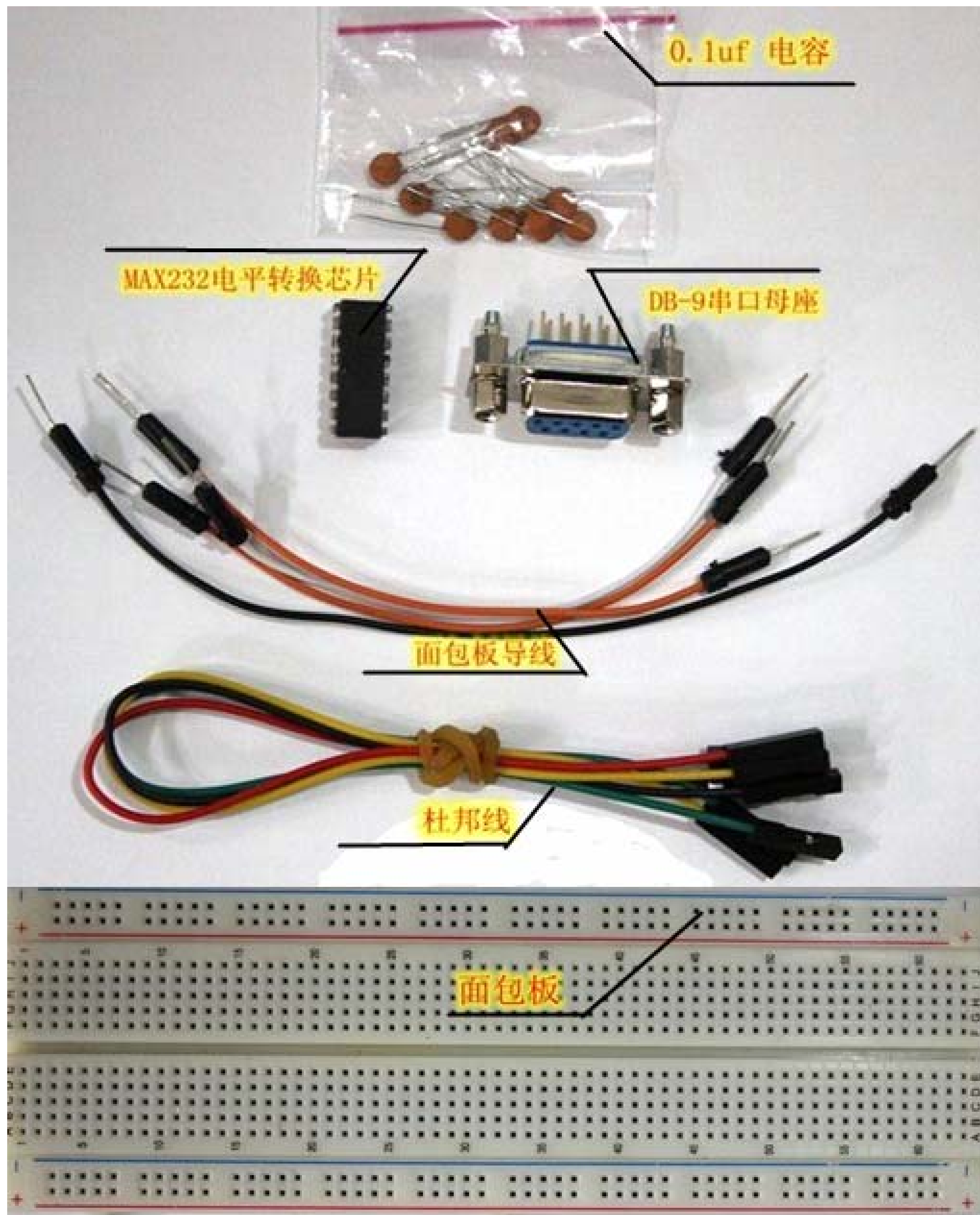


图 1-16 制作串口下载电路的材料

制作串口下载电路所需要的材料是在上文中面包板实验材料的基础上增加的，但并不多，也只有 5 种而已。下面表 1-5 列出他们的型号、数量和参考价格，购买的时候会很方便。

表 1-5 制作串口下载器的元器件清单

元 器 件 清 单				
品名	型号	参考价格	数量（个）	备注
电平转换芯片	MAX232	5.0	1	DIP 封装可用 MAX3232 代替
串口接头	DB9 母头	0.5	1	
电容	0.1uf	0.1	4	
导线	0.5mm	0.3	3	
面包板导线	2.54 间距	0.2	4	一捆几十条

MAX232 是一款常用的电平转换芯片，前面说了，它的功能是把 RS232 电平和 TTL 电平相互转换，它的外形与管脚数据手册如下图 1-17：

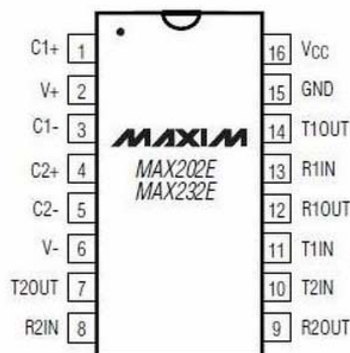


图 1-17 MAX232 芯片实物图与管脚图

我们的 PC 机串口输出的是由+12V 和-12V 的范围，输出的协议是一种叫 RS-232 的通信协议，而我们的单片机输出的是+5V 和 0V 的 TTL 电平，它也可以这种通信协议的，协议都相同但就是电压不相同，MAX232 就是解决它们电平不一致的问题.将电平相互转换而达成通信。其实还可以在网上找到许多种使用其他电路实现的电平转换，原理都是一样，我们这里只以 MAX232 为例，来做一根下载线出来，下图 1-18 是 MAX232 与 51 单片机芯片的连接原理图：

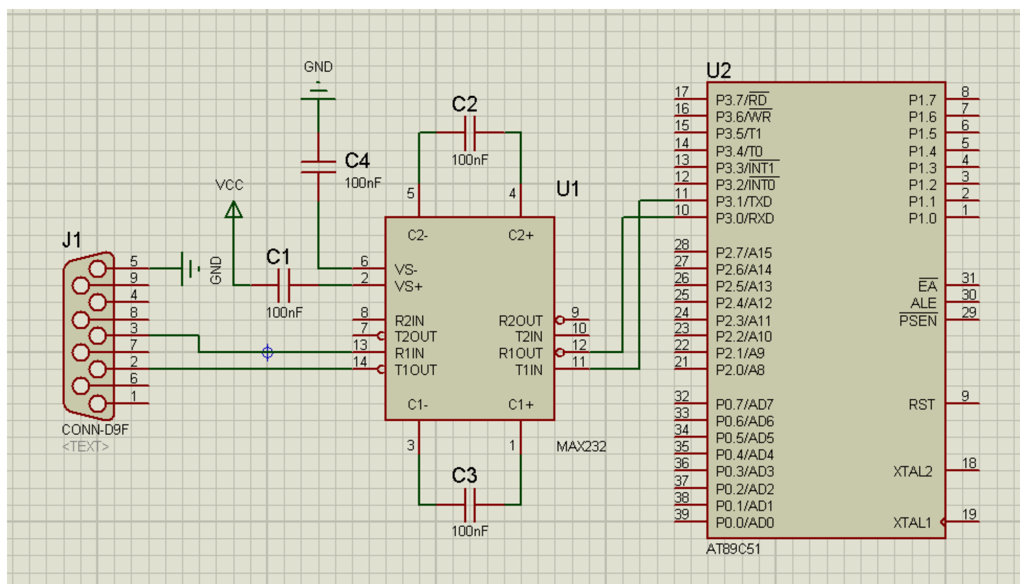


图 1-18 MAX232 与 STC 芯片的连接原理图

下面我把整个制作需要的实物以及制作过程都详细描述出来，大家可以跟着这些实物的照片一步一步的完成制作，然后有问题再回过头来研究一下原理图，这样会更快的学习到这些知识内容，接下来主要连接串口座的 GND，TX，RX 三根信号线，如下图 1-19 所示：



图 1-19 串口线三根主要连接线 GND，TX，RX

我们连线黄色线 RXD（PC 端串口接收），红色线 TXD（PC 端串口发送），蓝色线 GND（公共地），在座子上做好了几根不同颜色连接线如下图 1-20 所示：


图 1-20 杜邦线连接串口座

首先我们来制作 1 条 PC 机串口连接线，先将 PC 机主机箱上的串口连接到面包板上来，然后再进行进一步的电平转换。在 9 针串口接头上只需引出 2(TXD)、3 (RSD)、5(GND)共 3 根线就可以。使用面包板连接.可以方便地插入面包板，下表 1-6 列出了管脚连接顺序图。

表 1-6 电容连线说明

电容	MAX232
电容 1	1 脚
	3 脚
电容 2	4 脚
	5 脚
电容 3	2 脚
	16 脚
电容 4	6 脚
	15 脚
电容无正负极区分	

参照下图 1-21，可以把四只小电容插在芯片引脚对应的位置：

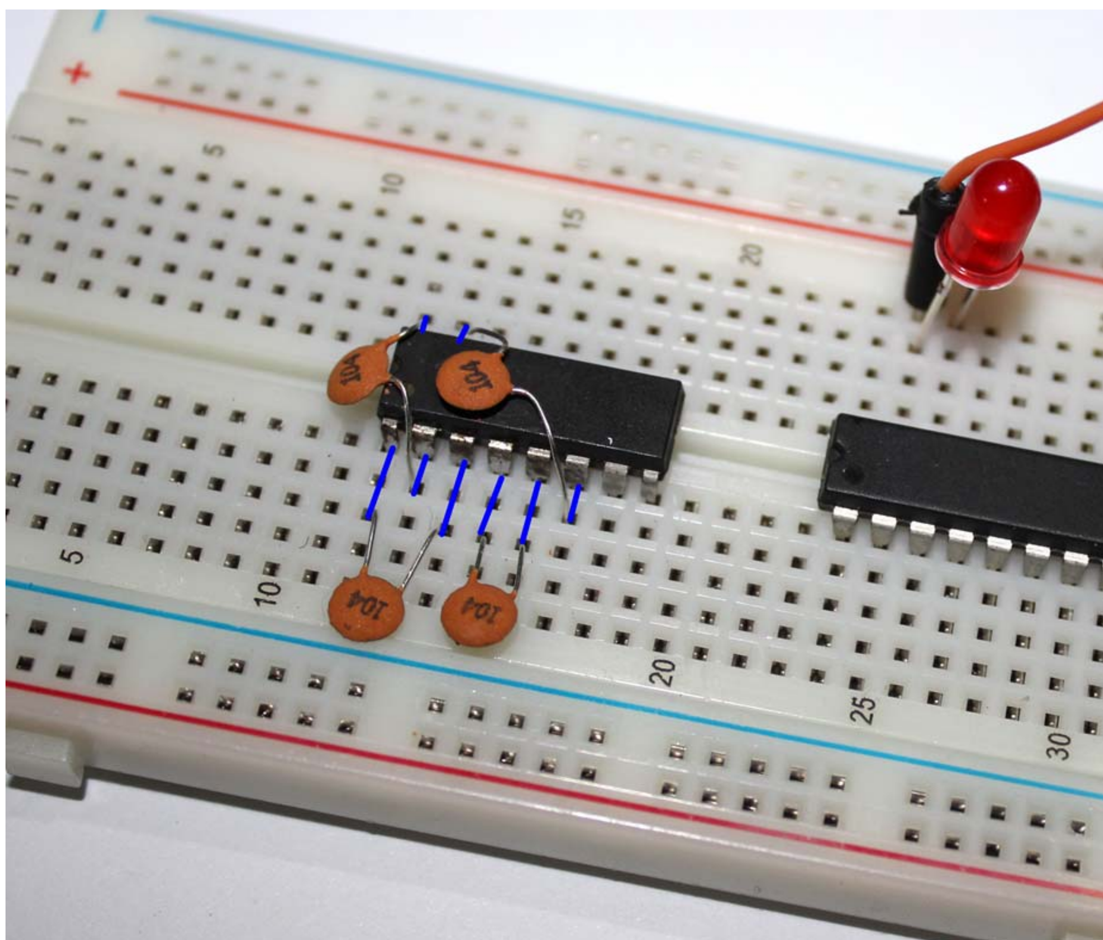


图 1-21 MAX232 与电容的连接图

接下来，开始连接串口线，将 MAX232 与串口线连接起来，连接的线序如下表 1-7：

表 1-7 MAX232 与串口连线对应表

串口连接线	MAX232
2（RXD）	14 脚
3（TXD）	13 脚
5（GND）	15 脚

下图 1-22 为连接好之后的实际效果图：

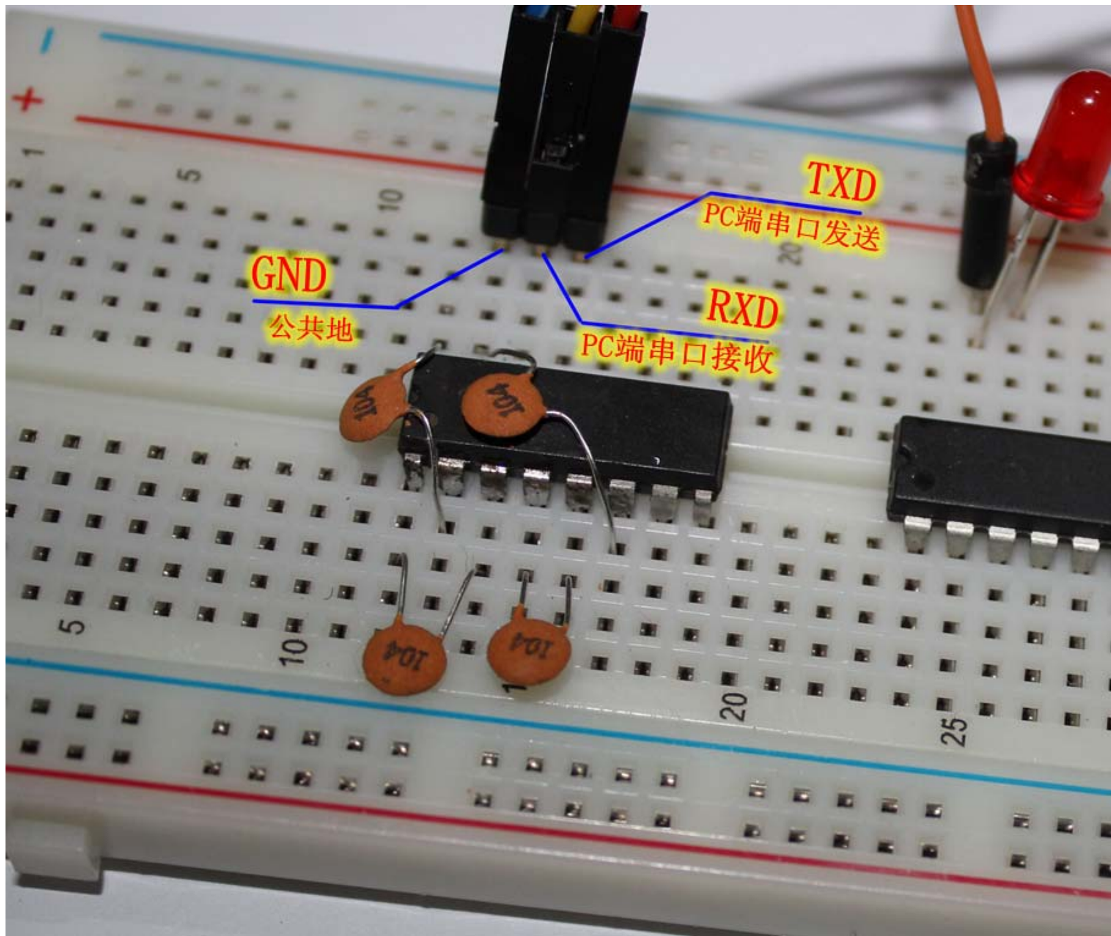


图 1-22 MA232 与串口线的连接图

下面开始更加复杂的操作，将 MAX232 与 51 单片机进行连接，连接的线序如下表 1-8：

表 1-8 串口转换芯片与 51 单片机连线说明

STC12C2052	MAX232	说明
2 脚	12 脚	
3 脚	11 脚	
20 脚	16 脚	电源 VCC
10 脚	15 脚	公共地 GND
用导线直接连接以上引脚		

下图 1-23 为连接好之后的实际效果图：

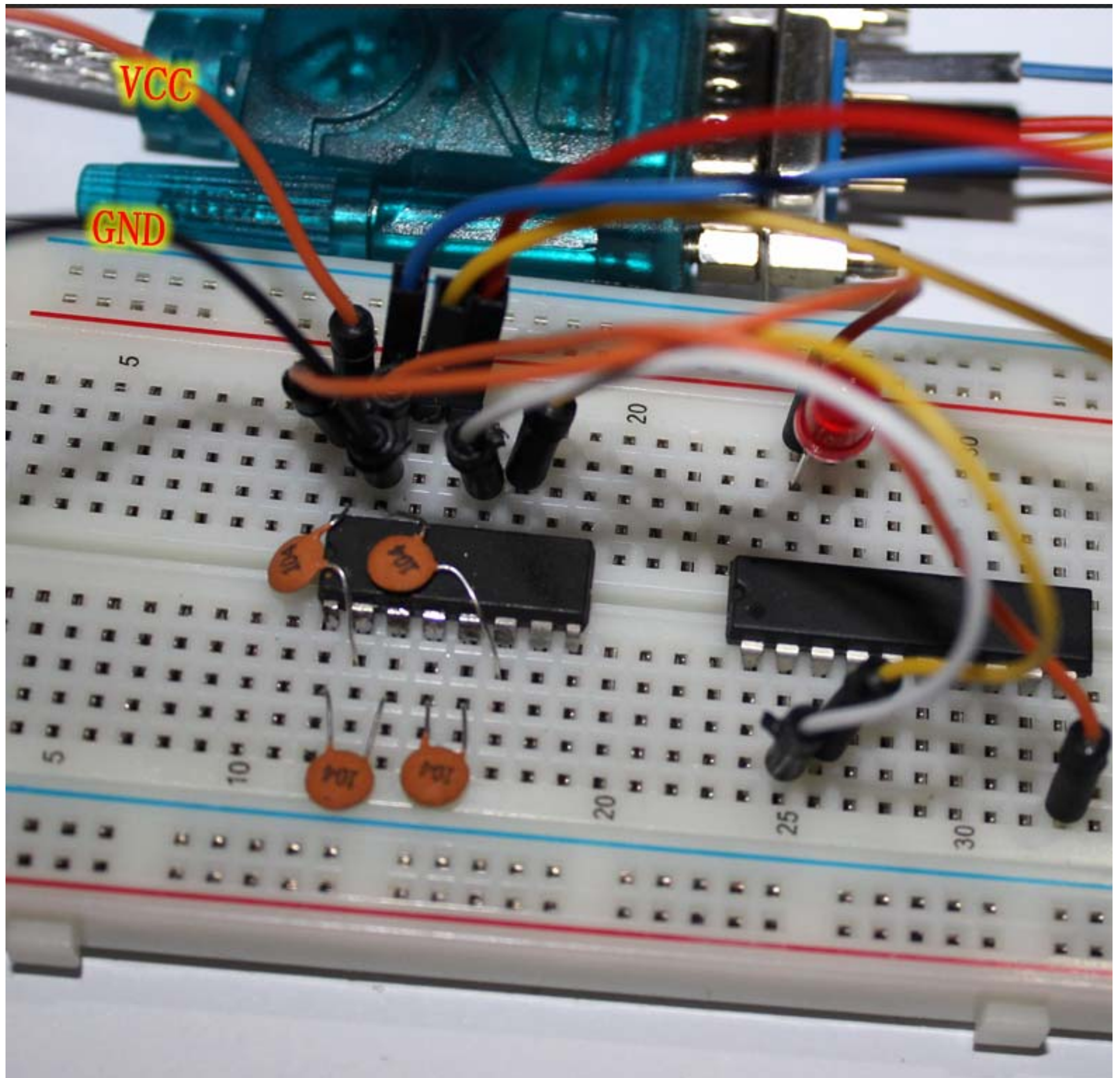


图 1-23 MAX232 与 51 单片机的连接图

在上文中提到的单片机实验电路里直接加入新的电路部分，把 MAX232 插在单片机的前面，然后按实物照片插接 4 个电容和串口连接线。接好后，将串口接头连接在 PC 机的串口上。或是 USB 转 RS-232 转换线上.如果电脑没有串口，可以在网上自己买一个 USB 转串口模块，将 USB 转串口线或 USB 转串口模块插入到电脑中后，如下图就会在【计算机管理】→【设备管理器】中出现一个新的串口设备，如下图 1-24 的红框所示：

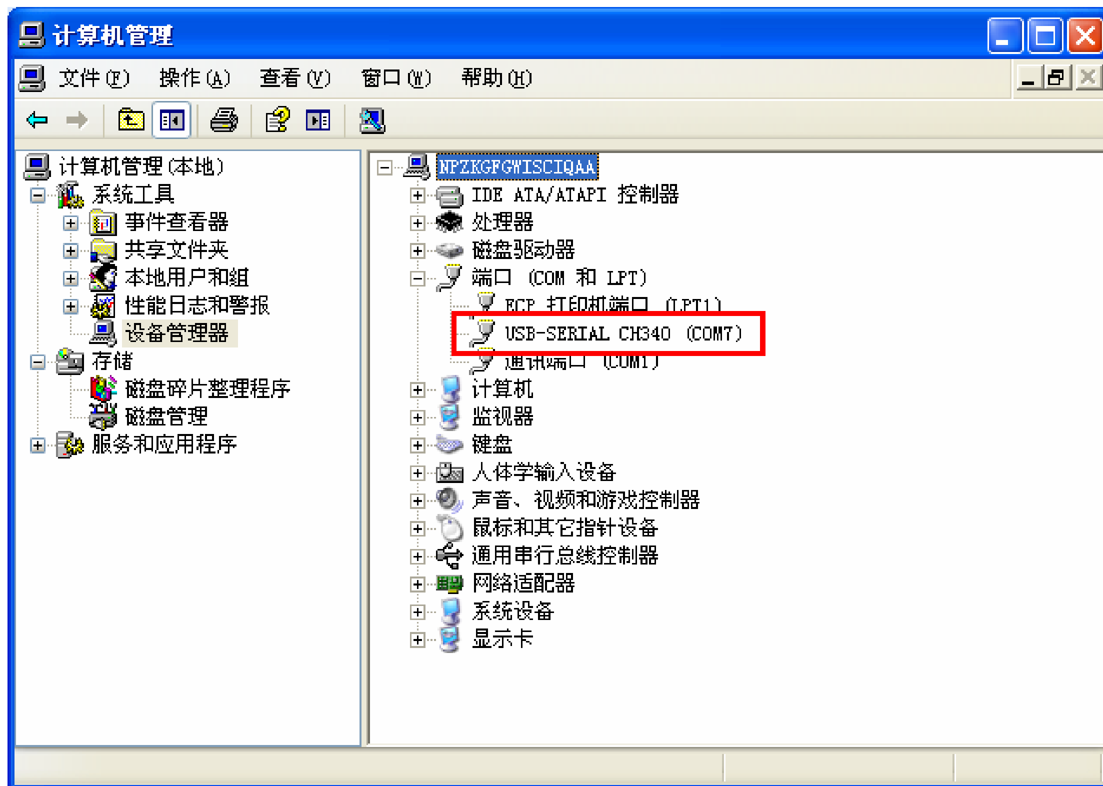


图 1-24 USB 转串口线插入后出现的新设备

建议你在 PC 机上使用 Windows XP 操作系统。其他的操作系统可能会出现设置方法不同或是不兼容等问题。Windows 的【开始菜单】→【控制面板】→【性能和维护】→【系统】→【硬件】→【设备管理器】→【端口（COM 和 TPL）】，可以看到我们 USB 转串口的设备 CH340，记住后面括号中出现的串口号（这个串口号是根据你接的 USB 口位置设定的）我们这里出现的是 COM5，在后面 51 单片机使用该串口下载的时候需要选择这个串口的号。

下面准备好之后开始打开下图 51 单片机的下载软件开始下载，双击图 1-25 这个图标：



图 1-25 STC 官方提供的下载软件

打开这个软件，如下图 1-26 所示：

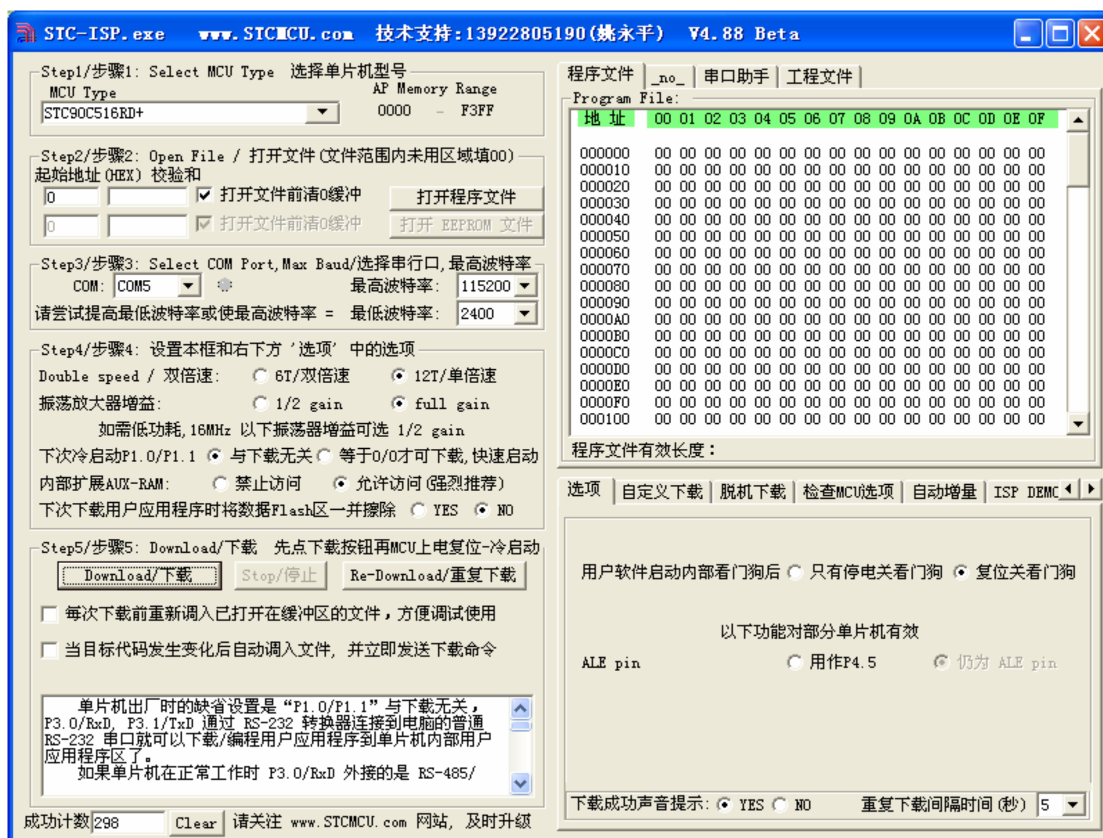


图 1-26 STC 下载软件界面图

可以看到这个软件上面标注着 5 个步骤，我逐个列一下：

在【Step1/步骤 1】中的单片机型号列表中选择 STC12C2052 型号，如下图 1-27 所示：

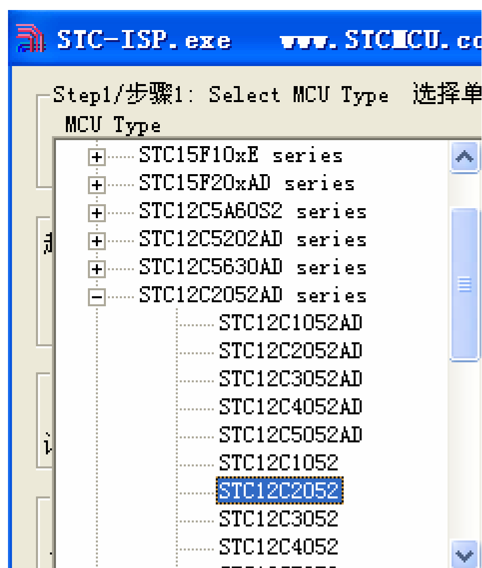


图 1-27 选择 51 单片机的型号

在【Step2/步骤 2】中点击【打开程序文件】，选择到打开已下载的本书配套资料，选择“步骤 6 51 单片机自己动手搭建最小电路\1.流水灯(神舟 51+ARM 之 DIY 自己动手搭建电路篇)\流水灯.hex 文件”，该文件由 KEIL 编译产生，如果没有找到文件，可以自行编译产生出来，如图 1-28：

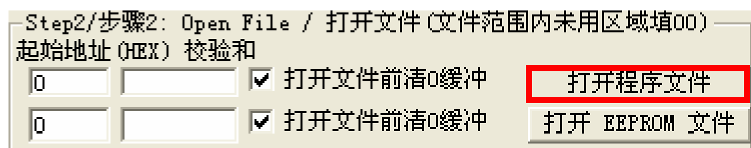


图 1-28 在 STC 软件中打开一个 HEX 文件

在【Step3/步骤 3】中选择串行口区域中选择在设备管理器中显示的串口号（刚才显示的是 COM7），波特率我们调节为最高最低都为 4800（这里值得注意的是如果波特率调得过高，会导致下载不成功），设置好之后如下图 1-29：



图 1-29 串口波特率设置图

在【Step4/步骤 4】中选择【内部 RC 振荡器】和【与下载无关】、【NO】这三个选择，如下图 1-30 的设置：

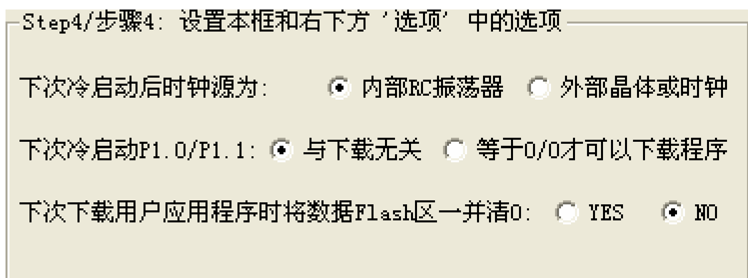


图 1-30 STC 软件配置

在【Step5/步骤 5】中点【Download 下载】按钮，建议勾上【每次下载前重新调入已打开在缓冲区的文件，方便调试使用】，这样在调试过程中，虽然程序重新编译了，但这个软件会重新载入最新编译出的 HEX 文件，具体如下图 1-31：

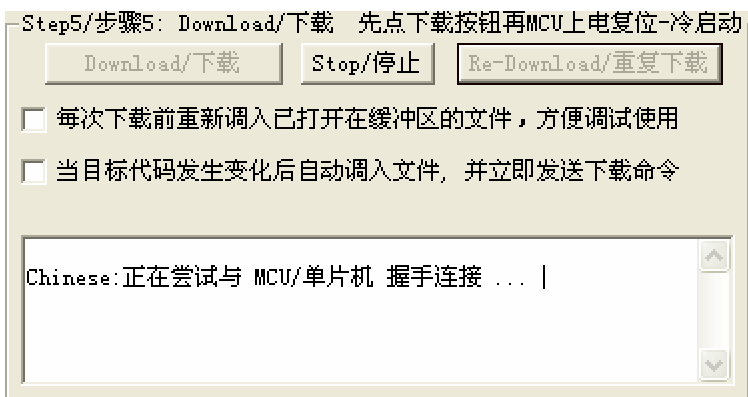


图 1-31 STC 软件下载设置

这时窗口左下方的状态窗口内显示【正在尝试与单片机握手连接…】，如下图 1-32：

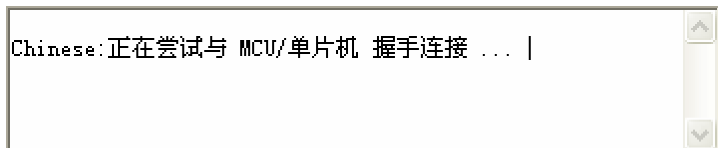


图 1-32 STC 软件正在等待握手

这表示 PC 机端已经准备就绪，正在等待单片机端的回应，这就好比两个人拍拖：

PC 机:51 单片机，你在吗，我有事找你，有话想跟你说呢，你在了回应我哦
单片机：PC 机你好，收到你的信息了，我在呀，有什么事情你说吧
PC 机：单片机，我喜欢你，做我女朋友吧，我有个程序要发给你
单片机：嗯，那我们沟通一下吧，接收你的程序。

当然有可能不是一帆风顺，如果 51 单片机一直没有回复，PC 机就会一直等待，那如果遇到一直没有回复的情况该如何处理呢？因为是一个冷启动的,这个 STC 的单片机需要断电然后再重新上电才可以下载的，这是因为厂商制作的引导程序就是这样的，只有单片机重新上电启动的时候，它的引导程序都会重新运行一次。

只有引导程序运行了，才有可能下载；其次还要看一下管脚线序是否连接对了，只有连接对了才有可能握手，完成下载，下图 1-33 表示已经下载成功！

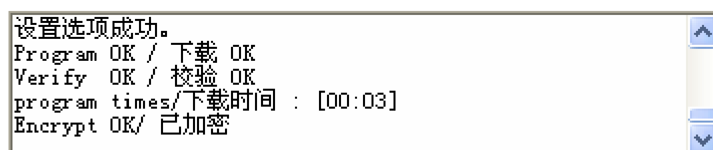


图 1-33 STC 软件下载成功的截图

1.5 制作USB下载线

1.5.1 关于USB下载的概念

USB 是英文 Universal Serial BUS（通用串行总线）的缩写，而其中文简称为“通串线，是一个外部总线标准，用于规范电脑与外部设备的连接和通讯。是应用在 PC 领域的接口技术。USB ,是英文 Universal Serial BUS（通用串行总线）的缩写，而其中文简称为“通串线，是一个外部总线标准，用于规范电脑与外部设备的连接和通讯。是应用在 PC 领域的接口技术。

用 USB 接口下载，其实只是一个 USB 转串口方式，也就是说 51 单片机芯片只有串口，无论 PC 机的接口是什么，最终都要转成串口的方式才能下载程序到单片机里，我们这里打算使用 USB，所以也可以直接使用 USB 转 TTL 的模块来代替，网上很多卖的，价格很便宜。

这里使用的是 USB 转 TTL 模块，这个模块也是 STM32 神舟系列厂家制作和生产的，它自带有 5V 和 3.3V 电源输出，可以直接给 5V 或 3V 的单片机供电，可以省去电池盒。

说到 USB 转 TTL 模块与串口有什么关系呢？有些朋友可能不太明白,简单的说单片机的串口具有 TTL 电平的输入/输出，通信的协议串口标准，可是电平并不是+12V、-12V，而是单片机上的 TTL 电平。PC 机的串口 RS-232 是+12V~-12V 的电平，这个需要用 MAX232 芯片才转换成标准的 TTL 电平，才能与单片机相连。如果还是不理解也没关系，后面我们会有更多的详细例程，慢慢就会明白。

1.5.2 用USB转TTL模块接口下载程序

首先准备好各种工具和组件，如下图 1-34：

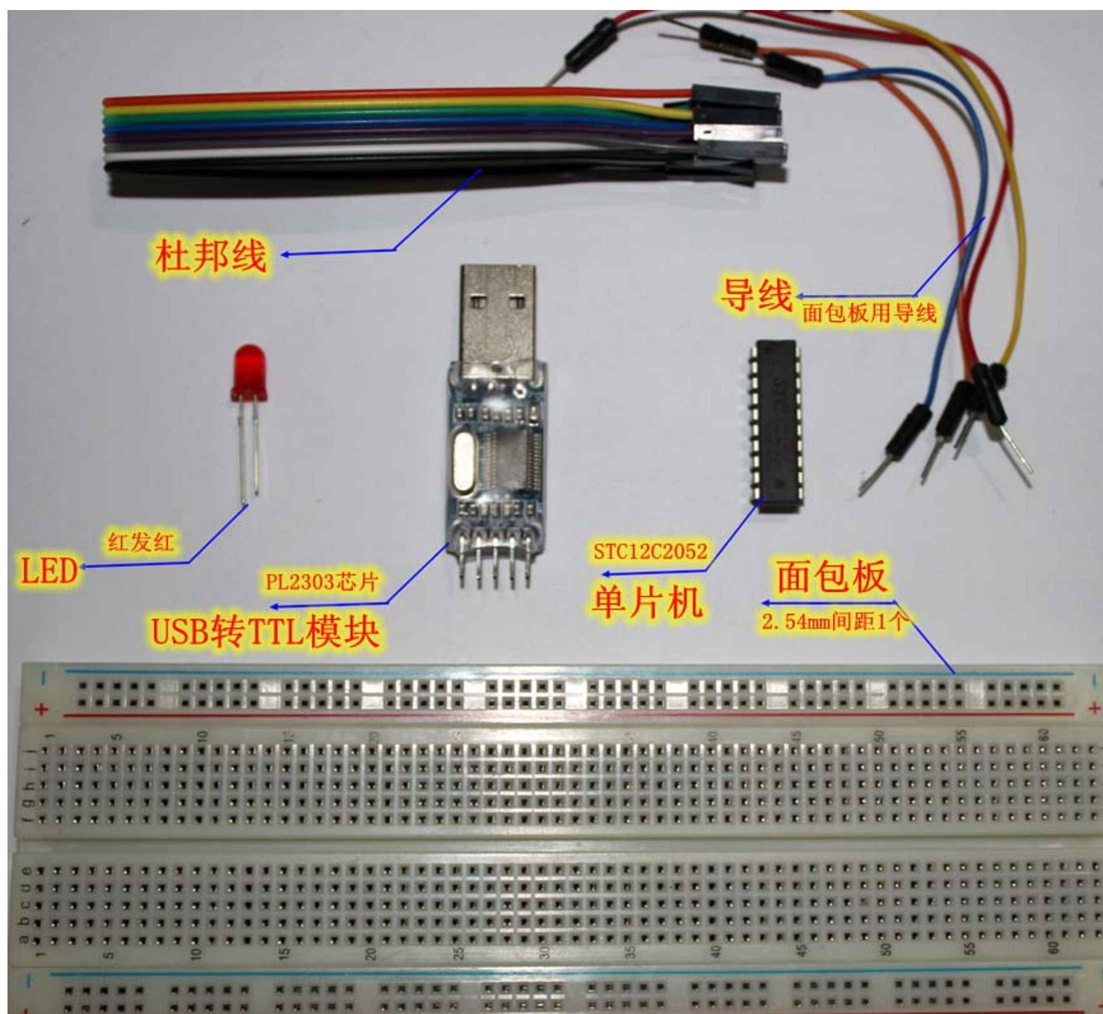


图 1-34 USB 转 TTL 下载模块器材准备

元器件清楚请参考表 1-9 所示：

表 1-9 下载模块元器件清单

品名	型号	参考价格（元）	数量(个）	备注
USB 转 TTL 模块		4.8	1	PL-2303HX 核心
单片机	STC12C2052	5.0	1	20 脚 DIP 封装
LED	直插 5mm	0.20	1	可选各种其他颜色和型号的 LED
杜邦线	0.1mm 长	2.0	4	
面包板板	2.54mm 间距	10.0	1	
面包板导线		0.10	4	一捆 70 条 10 元

下图 1-35 是 USB 转 TTL 模块与 51 单片机 STC12C2052 的连接原理图：

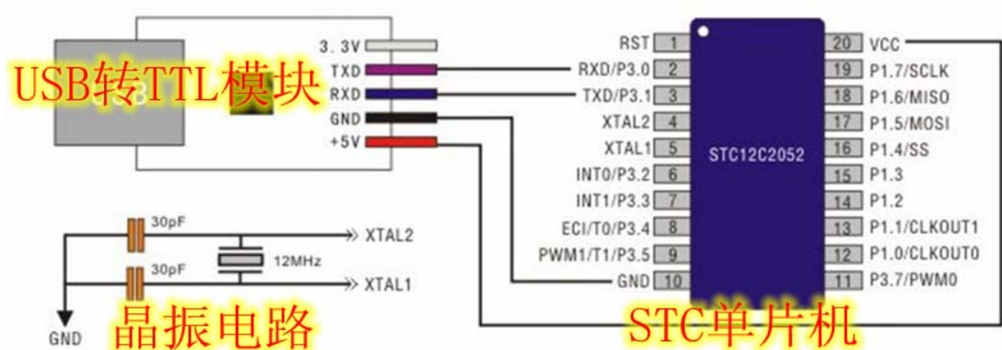


图 1-35 USB 转 TTL 模块与 51 单片机原理图

下表 1-10 中列出了模块的管脚与 51 单片机管脚一一对应表：

表 1-10 模块与 51 单片机芯片管脚对应列表

USB 转 TTL 模块	51 单片机（STC12C2052）
TXD	2 脚
RXD	3 脚
GND	10 脚
+5V	20 脚

根据原理图的标示，然后用导线直连以上的引脚，如下图 1-36：

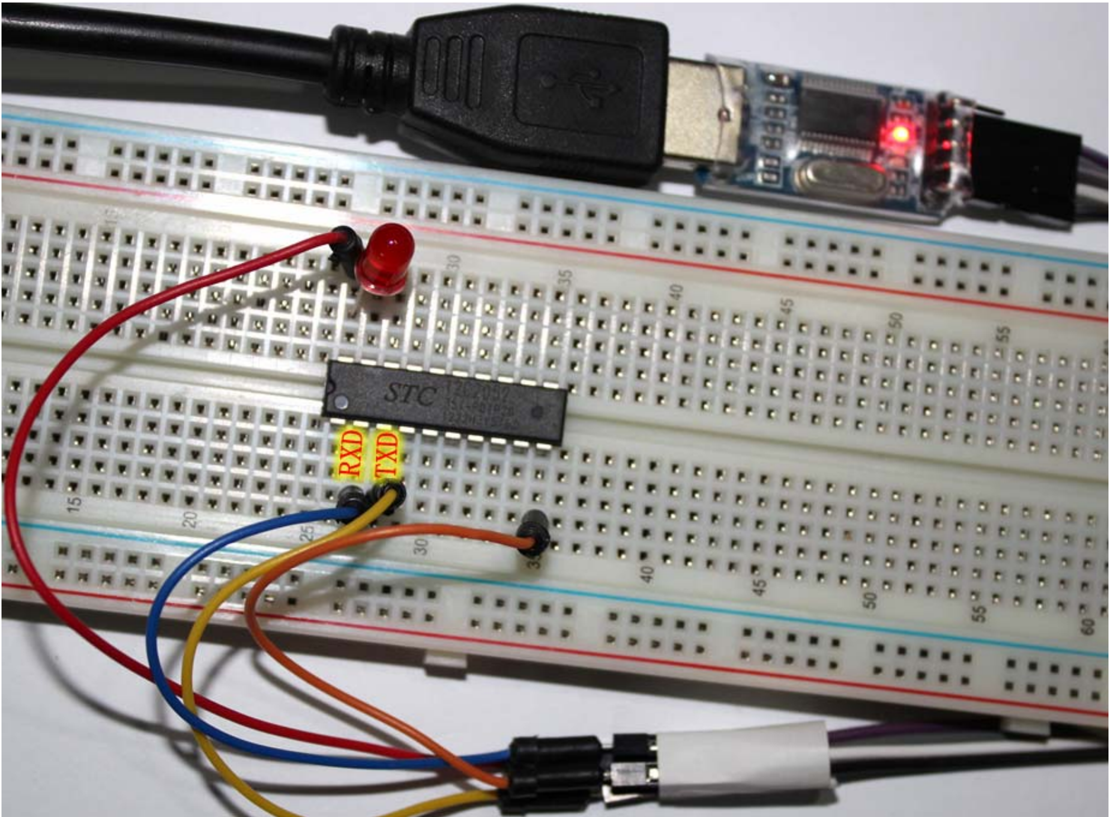


图 1-36 USB 转 TTL 模块连接 51 单片机管脚

使用 USB 下载需要把 PC USB 接口输出的电源转换成单片机可以使用的 TTL 电平，那么就要做一根专用的 USB 下载线。USB 转 TTL 模块可以把 PC USB 接口输出的电源转换成单片

机使用的电平，所以制作 USB 下载线选择一个 USB 转 TTL 模块和四根杜邦线就可以制作。

用 4 种不同颜色的杜邦线连接上 USB 转 TTL 模块的引脚（分别为红，蓝，黑，白四种颜色），如下图 1-37 所示：



图 1-37 USB 转 TTL 模块与 4 种颜色的杜邦线进行连接

连好线之后，把模块的一端 USB 座子插入到电脑中，这个时候，我们可以通过电脑的设备管理器看到这个模块被识别到了,接下来我们就通过这个端口下载我们的程序到我们的单片机中实现我们要的效果。

制作完成开始下载一个程序试试,准备好之后开始打开 51 单片机的下载软件开始下载。

1.6 自己搭建流水灯

1.6.1 实验说明

继续深入学习，这次实验搭建多个 LED 灯，我们使用 USB 下载线为单片机供电，电路也得到了很大的简化，USB 接口标准就是 5V 的电压输入的，这次我们把晶振电路加上，虽然单片机内部集成了一个不太精准的晶振电路，但是接个外部晶振时钟供给更加精准，外部晶振如果坏了，单片机还可以自动切换到内部进行运转。

下面先介绍一下晶振电路，晶振的学名叫晶体振荡器，它的种类很多，最常见的是石英晶体振荡器，它的振荡频率在 0~100MHz 之间。晶振电路的功能简单来说就是在电路中产生稳定的振荡频率，单片机可以使用这个频率作为自己的时间基准。相当于给单片机一个时间基准，让它很有时间观念，可以精确计算时间，按照固定的节奏一步一步处理我们交给它的任务；普通的晶体振荡器，旁边还需要使用 2 颗 30pF 的电容，帮助它起振和达到精确度。

普通的 LED 灯的工作电流在 10~30mA 众之间，如果超过低于这个值，有可能灯会不亮；如果超过这个值，灯可能会被烧坏，所以严格來讲，这里需要加一个合适的限流电阻，所以

在这个电路中每个灯上都串联了 1 只 100Ω的电阻，这是为了保护 LED 不被过高的电流烧坏而设计的。

电阻值计算公式是 R=（Vcc-Va）/I，其中 Vcc 为电源电压，Va 为 LED 正向驱动电压，I 为 LED 正向工作电流，假定我们所用的正向驱动电压 2V、工作电流希望保持 30mA、则 R=(5V-2V)/0.03A=100Ω。为什么在上一节中的实验里没有用限流电阻呢？不用限流电阻是不对的，因为不接限流电阻时，LED 和单片机都承受着较高的电流，短时间没有问题，你有没有注意到 LED 亮度过高呢？但如果长时间点亮，将有可能烧坏单片机，或者不烧坏也会减短单片机芯片的使用寿命。

1.6.2 实验原理图

自己搭建的流水灯，每个单片机的芯片管脚连接一个 LED 灯，每个 LED 灯连接一个电阻，详细的原理图如下图 1-38 所示：

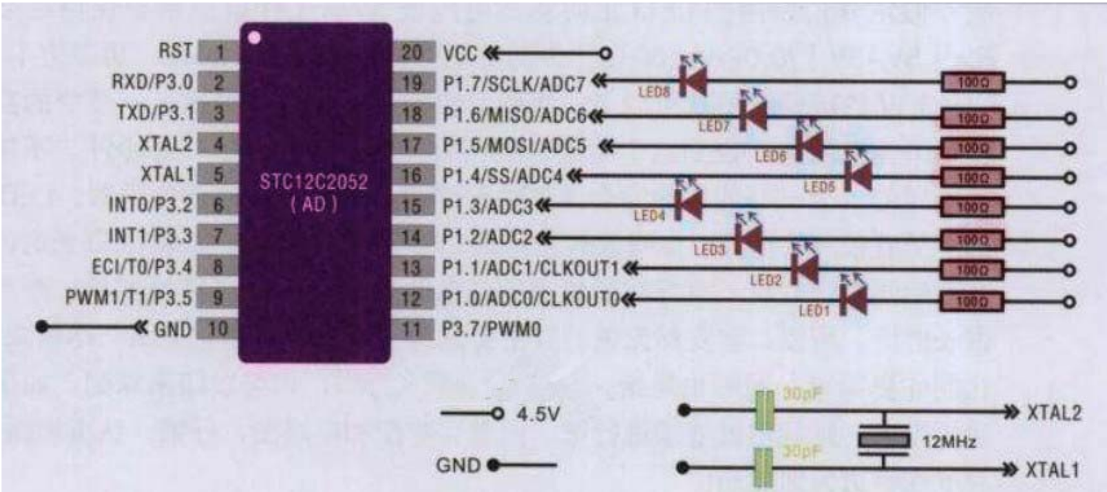


图 1-38 51 单片机流水灯电路原理图

1.6.3 器件清单与连接方法

下表 1-11 列出来 LED 流水灯搭建的器件清单，按照这个列表准备清单，就可以完成这个实验：

表 1-11 LED 流水灯搭建的器件清单：

序号	品名	型号	数量	参考价（元）	备注
1	电池盒	四节七号	1	2	保证输出电压在 4.5~5V
2	单片机	STC12C2052	1	5	可用 STC12C2052AD 替换
3	晶振	12MHz	1	0.8	
4	电容	30pF	2	0.01	陶瓷片电容
5	LED	直插 5mm	8	0.2	颜色型号不限
6	电阻	100Ω1/4W	8	0.01	
7	面包板	2.54mm 间距	1	10	颜色不限
8	面包板导线		15	0.01	一捆 10 元

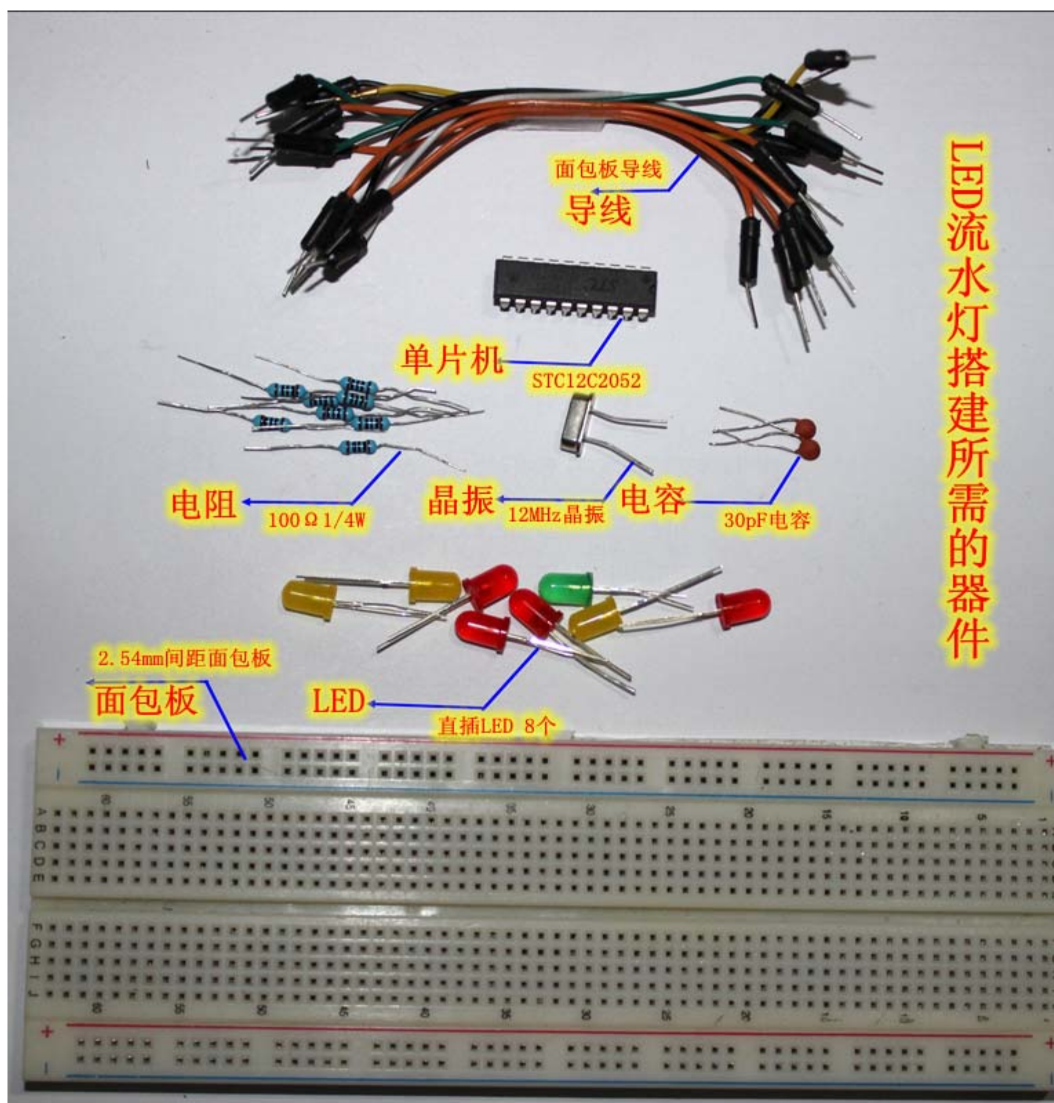


图 1-39 实验材料准备

根据原理连接图和器件清单，一一对应连接好之后如下图 1-40 所示：

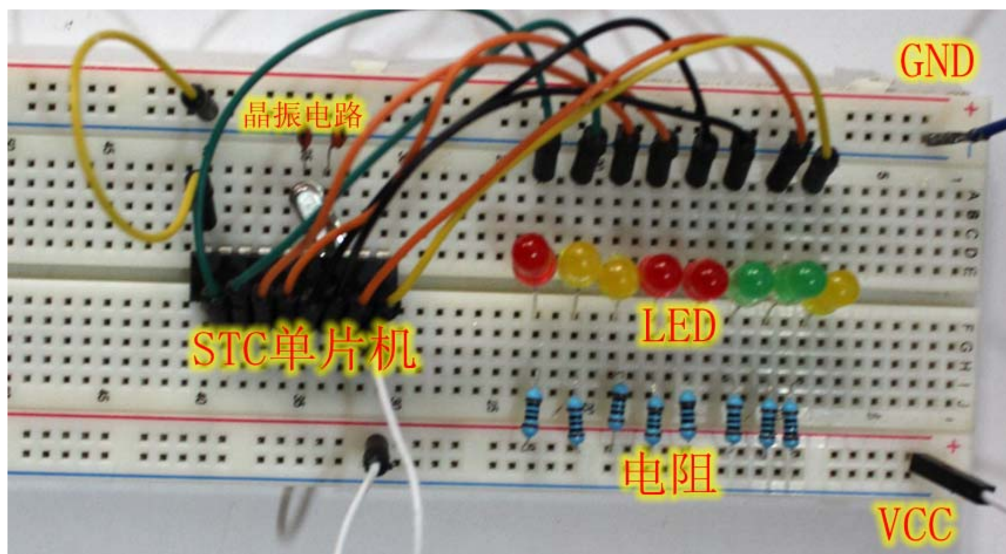


图 1-40 连接好 51 单片机点灯电路图

根据原理连接图和器件清单，一一对应连接好之后下图 1-41 是晶振电路的建立拍摄：

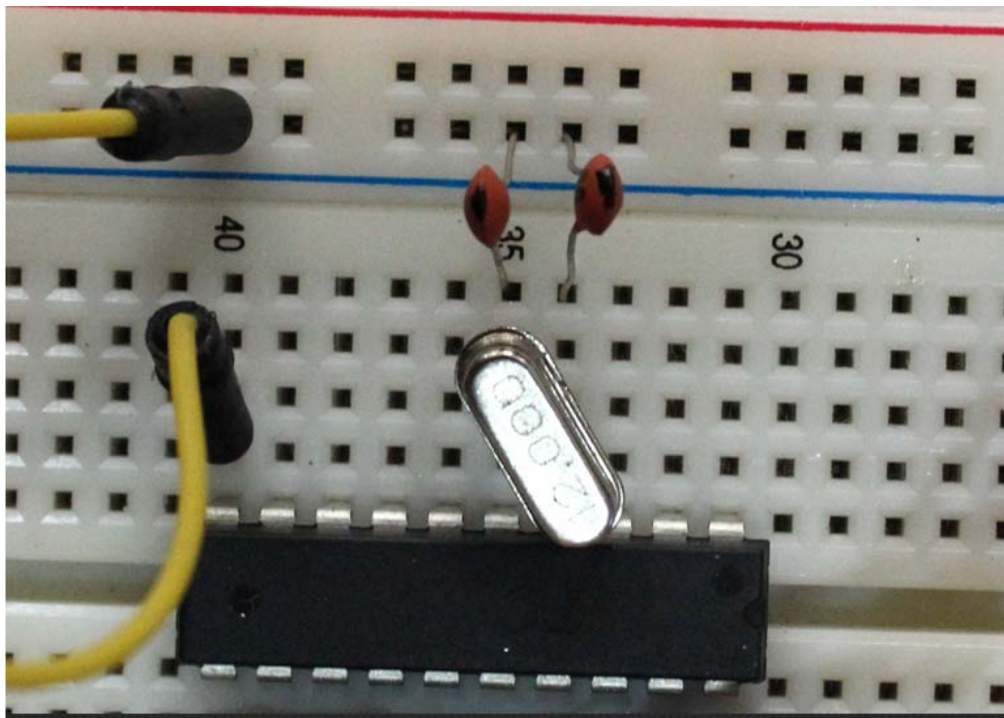


图 1-41 51 单片机的晶振电路

下图 1-42 是为 LED 灯插上限流电阻

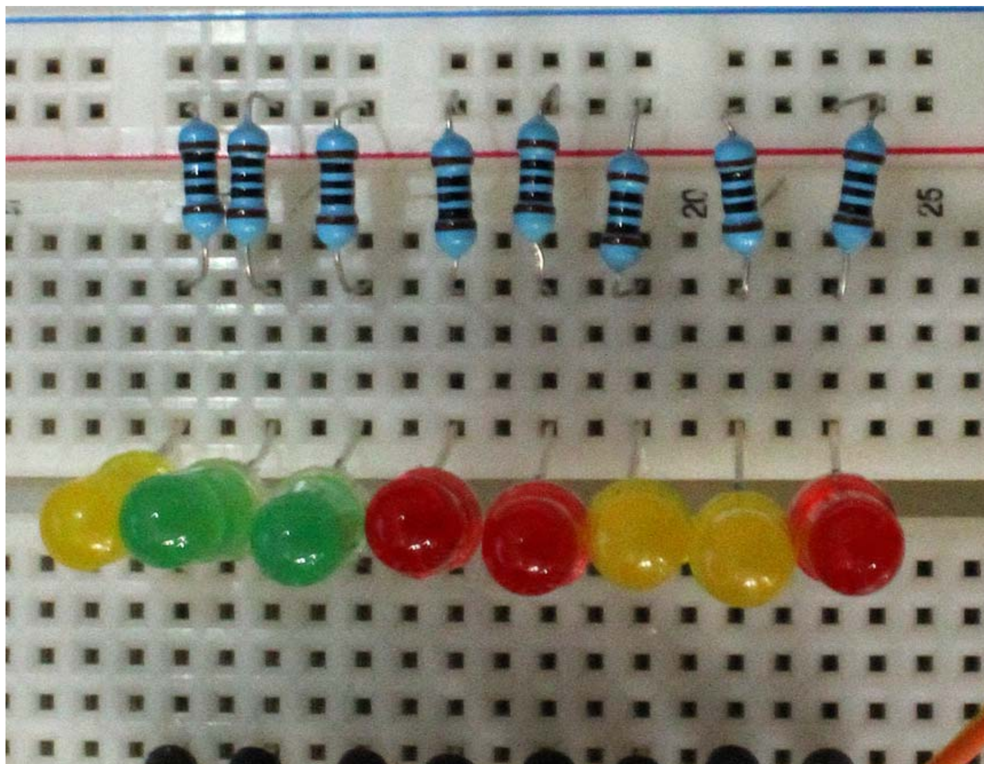


图 1-42 限流电阻

1.6.4 程序下载

打开已下载的本书配套资料，选择“步骤 6 51 单片机自己动手搭建最小电路\步骤 6 51 单片机自己动手搭建最小电路\1.流水灯(神舟 51+ARM 之 DIY 自己动手搭建电路篇)\流水灯.hex”文件，使用 STC 官方下载软件进行下载。

1.6.5 代码分析

程序中，我们调用了一个<reg.52>的一个头文件，在这个头文件里面为我们配置了各种寄存器等，在我们需要使用的时候就不用再次进行一个声明了，大大降低了我们的操作步骤，然后是我们的延时函数的声明，因为我们下面的程序中会使用到一个延时的效果，而这个延时函数的数值由用户进行配置，需要注意的是不能超过数据类型的最大值，在这个函数中，我们用到的是一个 unsigned int 的数据类型，即数值的配置在 0~65535 之间。代码如下：

```
#include<reg52.h> //包含头文件，头文件包含特殊功能寄存器的定义
void DELAY(unsigned int t); //函数声明
```

该延时函数如下，用户赋值到该函数中后，函数会对该值进行一个递减的过程，直到数值减为 0 的时候跳出该函数运行下面的程序。

```
/*------------------------------------------------
 延时函数，含有输入参数 unsigned int t，无返回值
 unsigned int 是定义无符号整形变量，其值的范围是 0~65535
------------------------------------------------*/
void DELAY(unsigned int t)
{
    while(--t);
}
```

下面我们来看下主函数是如何操作的，需要注意的是所有的程序都有一个主函数入口，并且也只能是有一个，程序由主函数入口开始运行。

函数中，我们先是声明了一个 unsigned char 型的 i 变量，数值范围在 0~255 之间，然后延时一段时间后给我们的 P1 口赋值 0xfe，这个 P1 在我们的头文件<reg52.h>中已经是声明好的了，这里我们直接使用就可以了，0xfe 为一个十六进制的数值，化成二进制为 1111 1110 也就是我们的 P1 的最低位为 0，其他都为 1，根据原理图可知，在 I/O 口上的 LED 为共阳级接法的，低电平点亮 LED，也就是说最低位的 LED 被点亮了。

赋值完 P1 口的值后，我们给它一个 for 循环，循环的次数我们设定为 8 次，循环中，每经过一段时间我们的 P1 口的值左移一位，比如开始的时候是 1111 1110 的，左移一位后就变成了 1111 1100，原来是点亮最低位的 LED 的，移位后变为点亮 2 个 LED，那么可得出移动 7 次后就能把 8 个 LED 全部点亮了，当循环完 8 次后跳出此循环，重新对 P1 口进行赋值，再次进入循环，达到我们想要的流水灯效果。

```
void main (void)
{
    unsigned char i;  //定义一个无符号字符型局部变量 i 取值范围 0~255
    DELAY(50000);
    P1=0xfe;            //赋初始值,0xfe 相当于二进制 1111 1110
    for(i=0;i<8;i++)    //加入 for 循环，表明 for 循环大括号的程序循环执行 8 次
```

```
    {
        DELAY(50000);
        P1<<=1;
    }

    while (0)           //主循环
    {
                //主循环中添加其他需要一直工作的程序
    }
}
```

1.6.5 实验现象

现在面包板上的 8 个 LED 在打开电源开关之后以一定速度来回亮着，就像击鼓传花来回不停的传，当你使用多种颜色的时，流动的效果会很精彩而漂亮，嘿嘿。如果这些灯按照有艺术的排列，比如排成一个心型或者其他形状，那就更加美观了。我记得大学的时候，有的同学自己制作了一个心形 LED 灯阵列向心爱的女生表白，好像后来他们成了，不知道跟这个有没有关系，所以大家一定要好好学会这个实验哦，下图 1-43 是效果图：

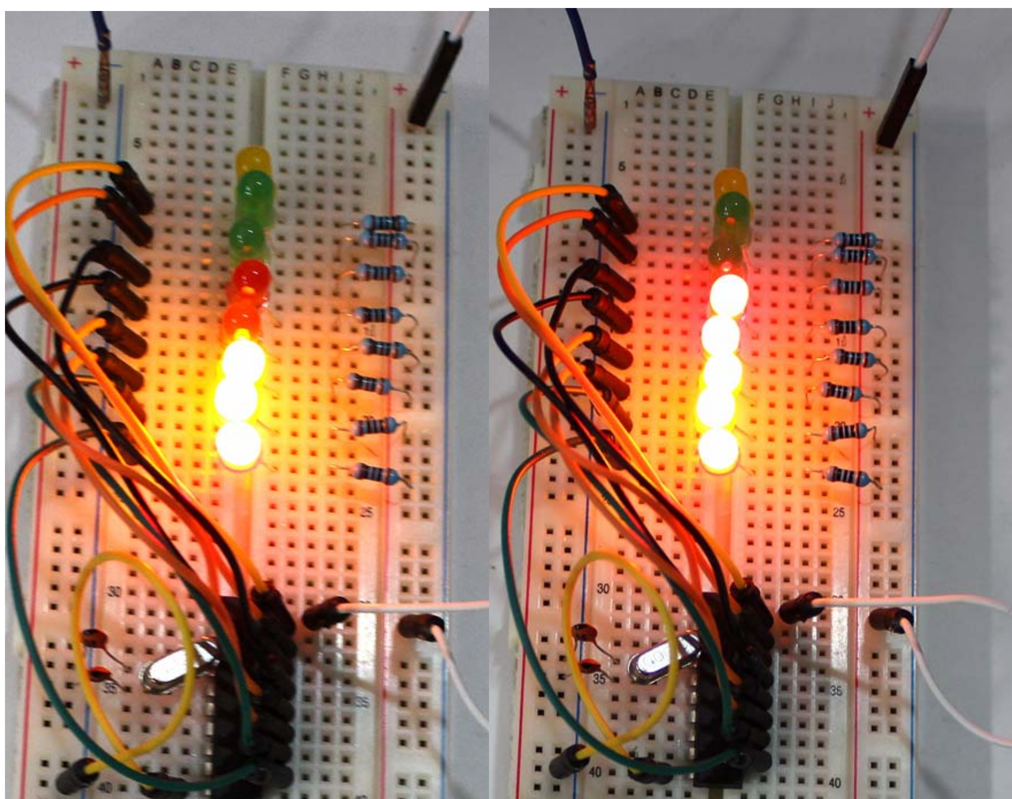


图 1-43 LED 流水灯效果图

1.7 按键控制LED灯

1.7.1 实验说明

用按键来控制 LED 灯，当我们按下按键的时候，单片机采集到这一个信号，使得按键去点亮 LED 灯，所有电路图请按照原理图进行实际的连接。

1.7.2 实验原理图

下面我们来看下 LED 部分的原理图与按键的实物结构图，其中图 1-44 为按键控制 LED 灯的原理图，图 1-45 为微动开关按键实物结构图：

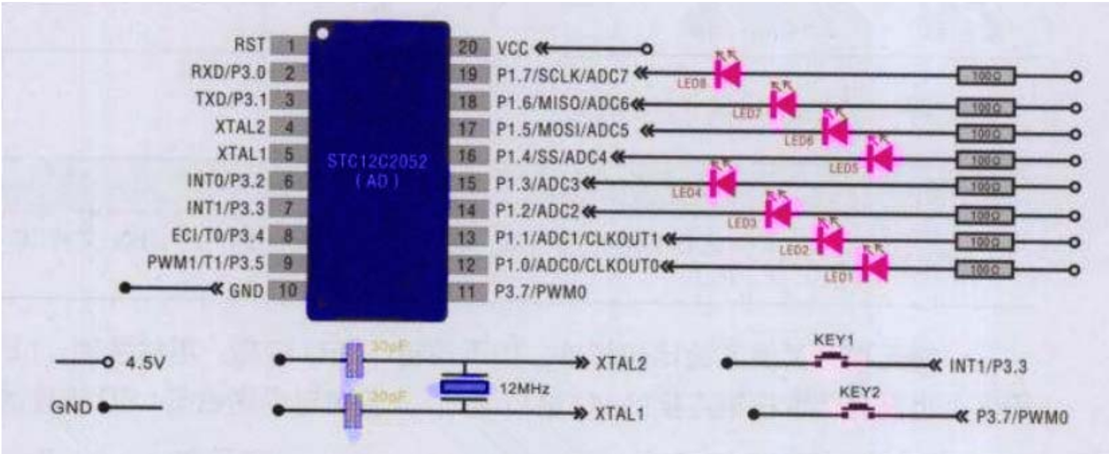


图 1-44 LED 电路原理图

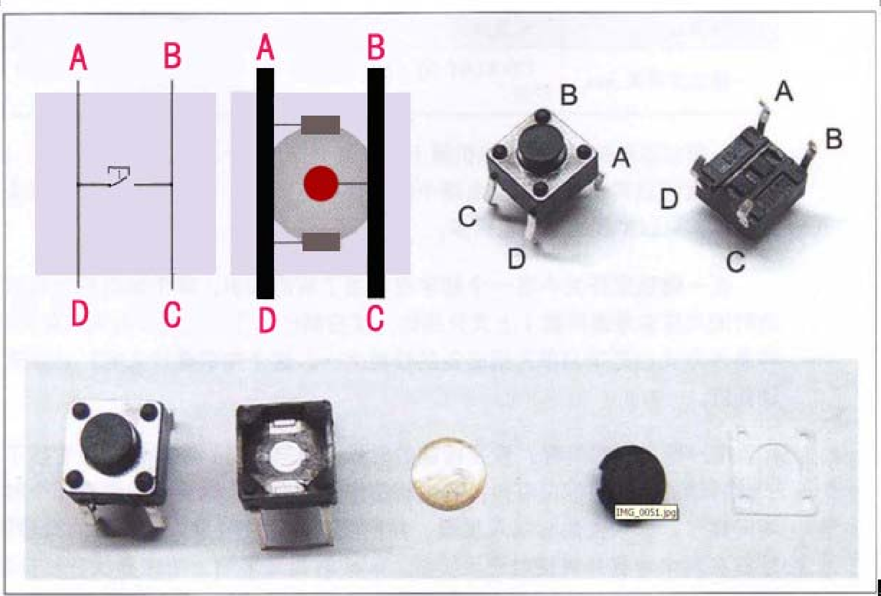


图 1-45 微动开关按键实物结构图

按照电路原理图把需要的材料准备好，如下图 1-46 为准备的材料：

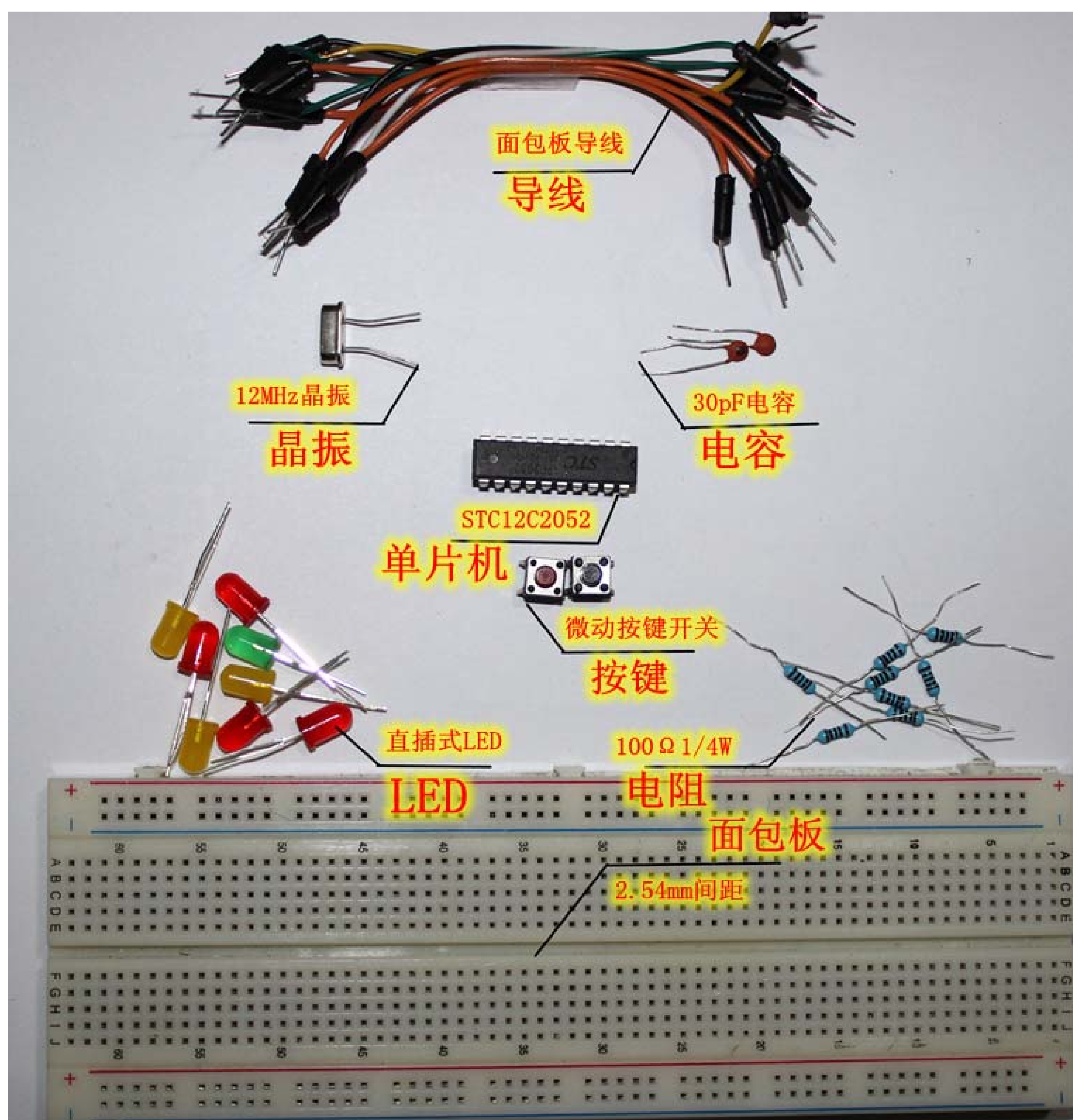


图 1-46 实验所需材料

按照电路原理图把以下电路连接好后如下图图 1-47、图 1-48、图 1-49 所示：

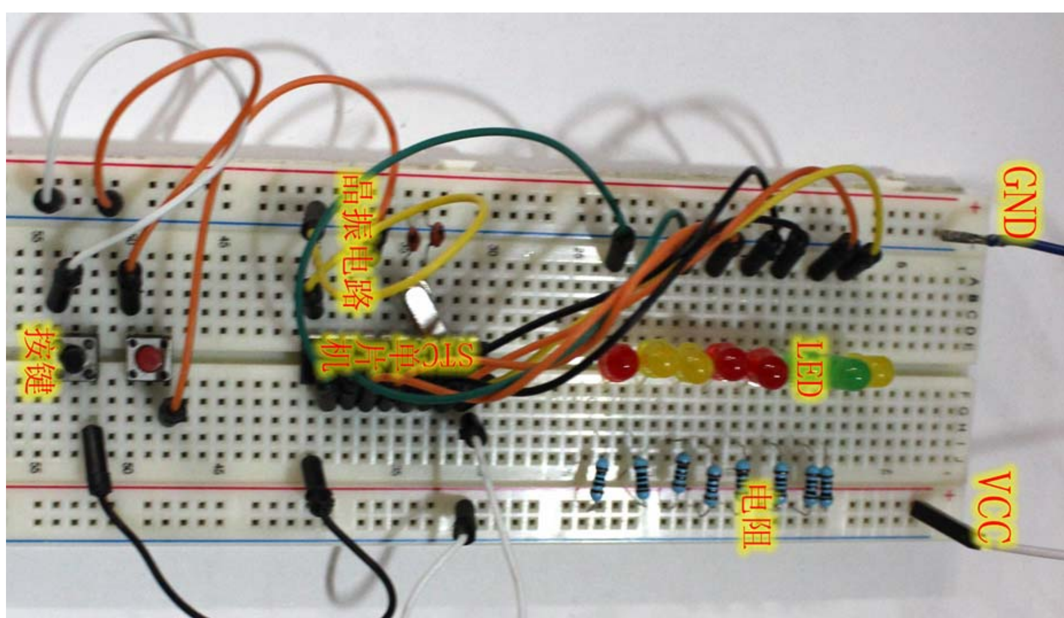


图 1-47 硬件环境连接图

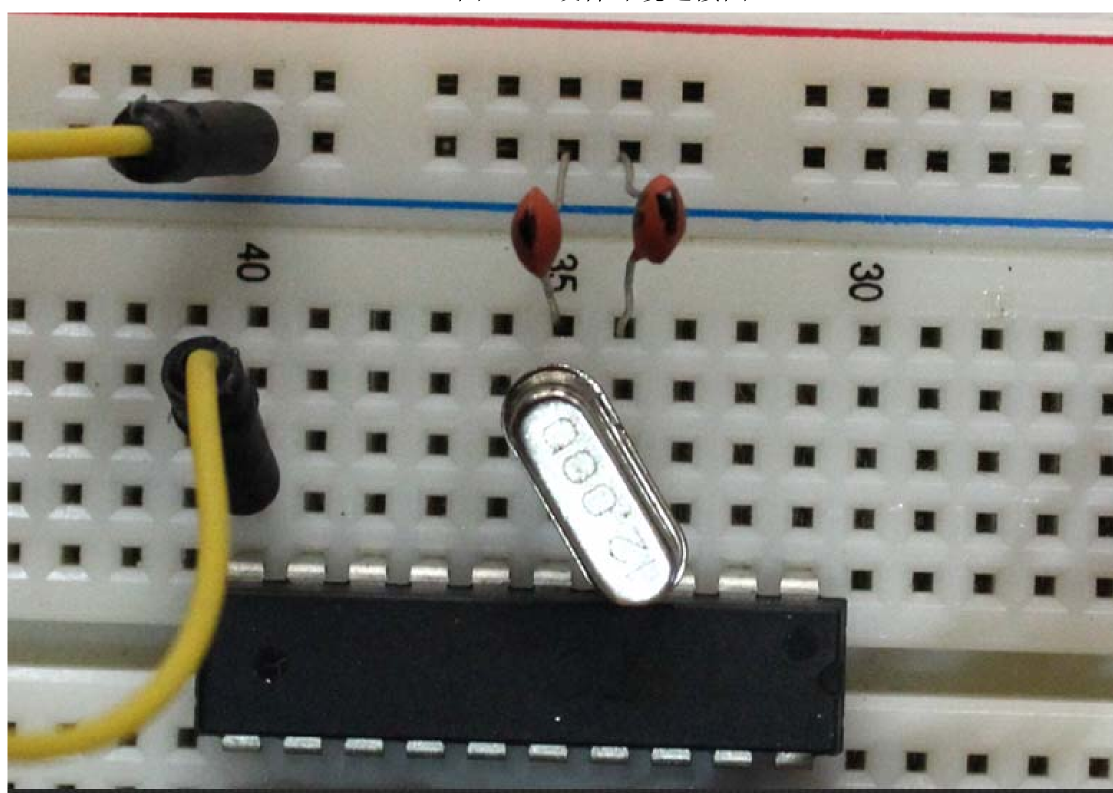


图 1-48 晶振电路

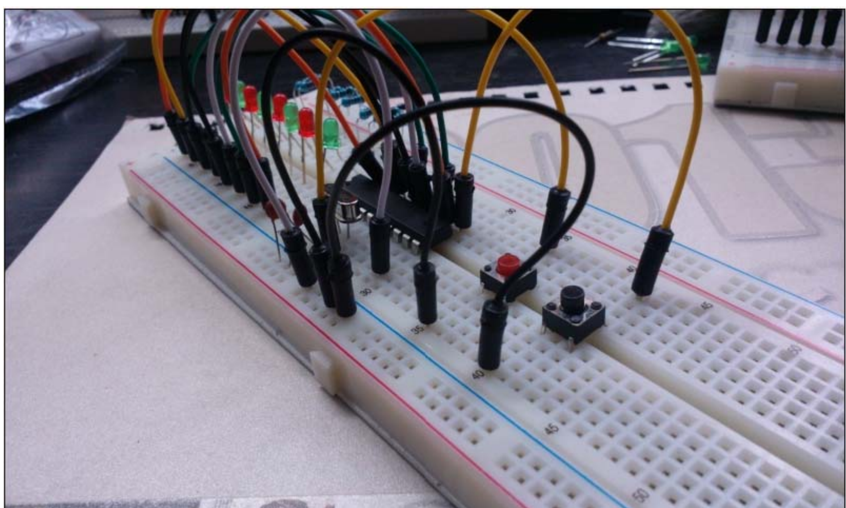


图 1-49 按键电路

1.7.3 器件清单与连接方法

我们这次实验所用到的器件材料如表 1-12 所示

表 1-12 按键控制 LED 灯实验器件清单

品名	型号	数量	参考单价（元）	备注
电池盒	四节七号	1	2	保证输出电压在 4.5~5V
单片机	STC12C2052	1	5	可用 STC12C2052AD 替换
晶振	12MHz	1	0.8	
电容	30pF	2	0.01	陶瓷片电容
LED	直插 5mm	8	0.2	颜色型号不限
微动开关	6mm*6mm*5mm	2	0.3	
电阻	100Ω1/4W	8	0.01	
面包板	2.54mm 间距	1	10	颜色不限
面包板导线		15	0.01	一捆 10 元

操作步骤：

1）了解面包板结构，分清各点之间的连接关系

2）查看电路图，点清配套元器件是否满足电路图所需元器件的数量

3）根据原理图在板子上搭建硬件环境，可参考实物图

4）检测搭建电路，可用万用表检测点与点之间的连接

5）烧录相对应的程序到单片机芯片上（配套的程序）

6）查看实验现象（按下一个按键点亮相对应的 4 个 LED 灯）

7）出现问题排查，检测元器件的好坏，对比原理图与所做板子的对比连接

1.7.4 程序下载

打开已下载的本书配套资料，选择“步骤 6 51 单片机自己动手搭建最小电路\步骤 6 51 单片机自己动手搭建最小电路\2.按键控制 LED 灯(神舟 51+ARM 之 DIY 自己动手搭建电路

篇)\按键与LED.hex”文件，使用STC官方下载软件进行下载。

1.7.5 代码分析

和前面的程序一样，调用了<reg52.h>的一个头文件，下来是我们的按键与LED的标识符声明，从这里我们知道了各个按键与LED连接到的I/O口位置，延时函数我们这里用到了2个，一个是微秒级的延时函数，一个是毫秒级的延时函数，毫秒级的延时函数要调用微秒级的延时函数才能达到微秒的延时。

```
/********************************************************
* 例程：单个独立按键检测
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：用于时刻检测按键状态并输出指示
*********************************************************/
#include<reg52.h> //包含头文件，一般情况不需要改动，头文件包含特殊功能寄存器的定义

sbit KEY1=P3^3;   //定义按键输入端口
sbit KEY2=P3^7;   //定义按键输入端口
sbit LED0=P1^0;   //定义 led 输出端口
sbit LED1=P1^1;   //定义 led 输出端口
sbit LED2=P1^2;   //定义 led 输出端口
sbit LED3=P1^3;   //定义 led 输出端口
sbit LED4=P1^4;   //定义 led 输出端口
sbit LED5=P1^5;   //定义 led 输出端口
sbit LED6=P1^6;   //定义 led 输出端口
sbit LED7=P1^7;   //定义 led 输出端口

void DelayUs2x(unsigned char t);
void DelayMs(unsigned char t);
/*-- 主函数 --*/
void main (void)
{
    KEY1=1; //按键输入端口电平置高
    KEY2=1; //按键输入端口电平置高
    while (1)          //主循环
  {
        if(!KEY1)     //如果检测到低电平，说明按键按下
            {
                LED0=0;
                LED1=0;
                LED2=0;
                LED3=0;
                DelayMs(100);
```

```
		}
	else
		{
		if(!KEY2)	//如果检测到低电平，说明按键按下
			{
			LED4=0;
			LED5=0;
			LED6=0;
			LED7=0;
			DelayMs(100);
			}
		}
	}
}
/*------------------------------------------------------------------------
 uS 延时函数，含有输入参数 unsigned char t，无返回值
 unsigned char 是定义无符号字符变量，其值的范围是
 0~255 这里使用晶振 12M，精确延时请使用汇编,大致延时
 长度如下 T=tx2+5 uS
------------------------------------------------------------------------*/
void DelayUs2x(unsigned char t)
{
	while(--t);
}
/*------------------------------------------------
 mS 延时函数，含有输入参数 unsigned char t，无返回值
 unsigned char 是定义无符号字符变量，其值的范围是
 0~255 这里使用晶振 12M，精确延时请使用汇编
------------------------------------------------*/
void DelayMs(unsigned char t)
{
	while(t--)
	{	//大致延时 1mS
		DelayUs2x(245);
		DelayUs2x(245);
	}
}
```

接下来我们看下主函数里面是怎么样的，函数中，我们先把按键的 I/O 口拉高，因为它是一个上拉输入的，只有按键按下去的时候才把 I/O 口由高电平拉为低电平，才检测到这个按键是已经按下去的了，接下来是我们的一个 while 大循环，通过判断语句 if 去判断哪个按键按下去了，按键按下去的时候是由 1 变为 0 的，也就是说在没按键按下去的时候，通过判断语句里面的条件取反后都不执行里面的代码（0 为假，其他为真，只有为真时才执行里面的函数，按键没按下去为 1，取反后为 0，不执行里面的程序）。当按键 KEY1 按下去的时候，0 取反为真，执行里面的程序，使 LED0~LED3 等于 0，LED4~LED7 等于 1,前面介绍

过，0 点亮，1 熄灭，所以我们点亮 LED0~LED3 而熄灭 LED4~LED7，并延时。这个就是按键 KEY1 所达到的效果，而 KEY2 和 KEY1 相反，它是按下去的时候使得 LED0~LED3 等于 1，LED4~LED7 等于 0，点亮 LED4~LED7 而熄灭 LED0~LED3.并延时

```
/*--  主函数  --*/
void main (void)
{
    KEY1=1; //按键输入端口电平置高
    KEY2=1; //按键输入端口电平置高
    while (1)           //主循环
  {
        if(!KEY1)      //如果检测到低电平，说明按键按下
        {
            LED0=0;
            LED1=0;
            LED2=0;
            LED3=0;
            DelayMs(100);
        }
        else
        {
            if(!KEY2)   //如果检测到低电平，说明按键按下
            {
                LED4=0;
                LED5=0;
                LED6=0;
                LED7=0;
                DelayMs(100);
            }
        }
    }
}
```

1.7.6 实验现象

通过 STC 官方下载软件把程序下载到我们的 STC 单片机芯片上后，给我们的面包板供电，这个时候你们会发现，当我们按下按键时 LED 点亮，不按按键 LED 不亮，如图 1-53 所示。具体 LED 和按键的控制可以在程序中改写控制哪一个 LED。

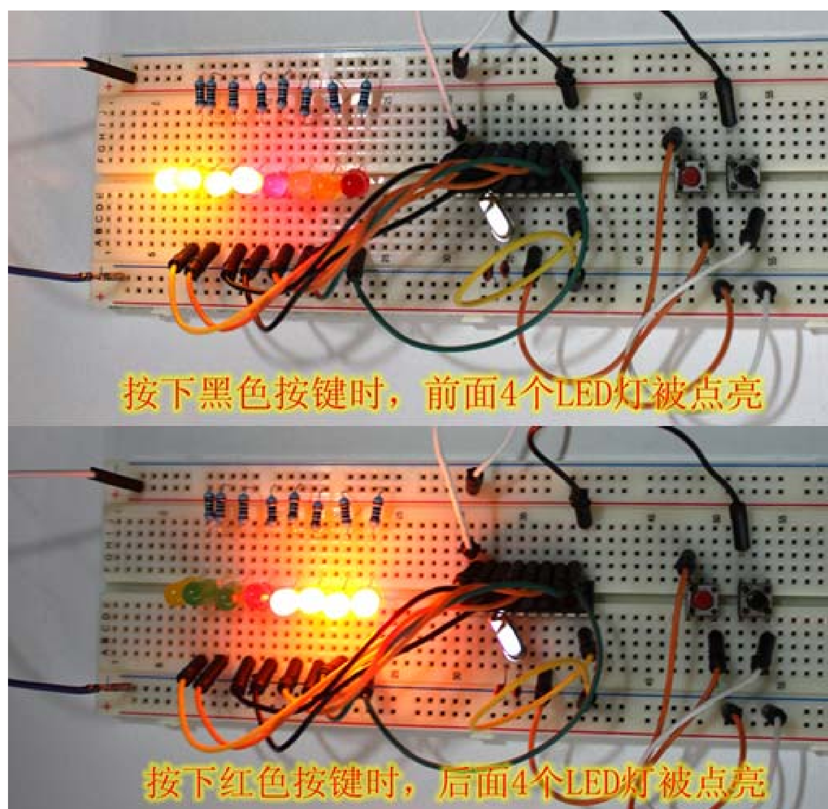


图 1-50 按键控制 LED 实验现象

1.8 按键控制蜂鸣器嘟嘟嘟的响

1.8.1 实验说明

使用该例程来使得按键按下，蜂鸣器响。

1.8.2 实验原理图

本次实验的原理图如图 1-51 所示，该部分为蜂鸣器电路图

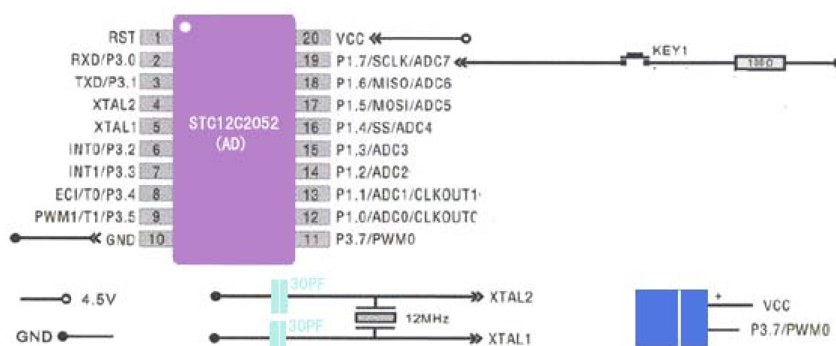


图 1-51 蜂鸣器实验原理图

1.8.3 器件清单与连接方法

本次实验所用到的器件材料如表 1-13 所示

表 1-13 按键控制蜂鸣器实验器件清单

品名	型号	数量	参考单价（元）	备注
电池盒	四节七号	1	2	保证输出电压在 4.5~5V
单片机	STC12C2052	1	5	可用 STC12C2052AD 替换
晶振	12MHz	1	0.8	
电容	30pF	2	0.01	陶瓷片电容
蜂鸣器	5V	1	1.0	无源蜂鸣器
微动开关	6mm*6mm*5mm	1	0.3	
电阻	100Ω1/4W	1	0.01	
面包板	2.54mm 间距	1	10	颜色不限
面包板导线		8	0.01	一捆 10 元

1）了解面包板结构，分清各点之间的连接关系

2）查看电路图，点清配套元器件是否满足电路图所需元器件的数量

3）根据原理图在板子上搭建硬件环境，可参考实物图

4）检测搭建电路，可用万用表检测点与点之间的连接

5）烧录相对应的程序到单片机芯片上（配套的程序）

6）查看实验现象（按键按下，蜂鸣器发出声音）

7）出现问题排查，检测元器件的好坏，对比原理图与所做板子的对比连接

根据所要用到的材料表格，配备的材料实物图如图 1-52 所示：

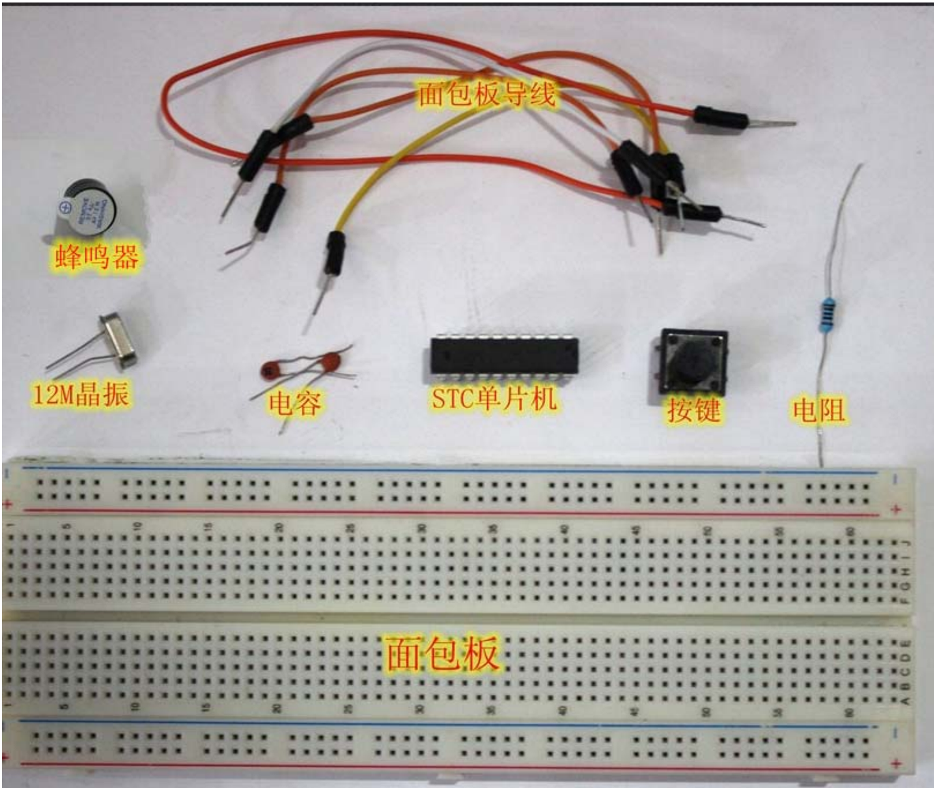


图 1-52 蜂鸣器实验所需材料

根据电路图，我们在面包板上搭建我们的电子线路，完成我们的这次实验，硬件环境搭建结

果如图 1-53 所示：

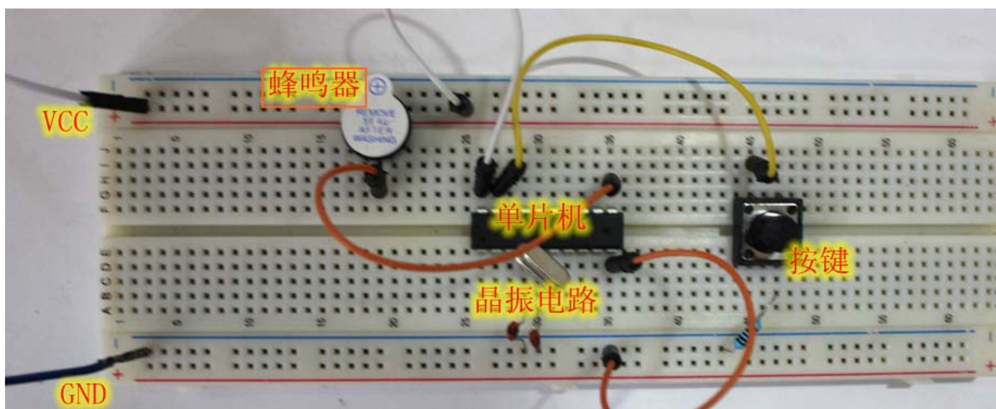


图 1-53 搭建电路结果

1.8.4 程序下载

打开已下载的本书配套资料，选择“步骤 6 51 单片机自己动手搭建最小电路\步骤 6 51 单片机自己动手搭建最小电路\3.按键与蜂鸣器(神舟 51+ARM 之 DIY 自己动手搭建电路篇)\按键与蜂鸣器.hex”文件，使用 STC 官方下载软件进行下载。

1.8.5 代码分析

本次的代码比较少，所以也比较容易理解，同理，先是调用头文件<reg52.h>，然后声明我们的按键与连接蜂鸣器的 I/O 口，主函数只有一个判断语句，判断该按键是否按下，当按键按下的时候让连接蜂鸣器的 I/O 口输出一个低电平，否则输出高电平，这样我们就能通过按键控制 I/O 口输出电平的变化控制我们的蜂鸣器了。

```
/*******************************************************
* 例程：单个独立按键检测
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：用于时刻检测按键状态并输出指示
*******************************************************/
#include<reg52.h> //包含头文件，一般情况不需要改动，
                  //头文件包含特殊功能寄存器的定义
sbit KEY=P1^7;    // 用 sbit 关键字 定义 LED 到 P2.0 端口?
sbit PWM=P3^7;   //LED 是自己任意定义且容易记忆的符号
/*------------------ 主函数 ------------------------*/
void main (void)
{
    if(KEY==0)
    PWM=0;
    else
    PWM=1;
```

```
}
```

1.8.6 实验现象

下载程序后的实验，如图 1-54 按键控制蜂鸣器实验现象所示，按下按键 I/0 接口产生高低电平变化，推动扬声器振动发出声音

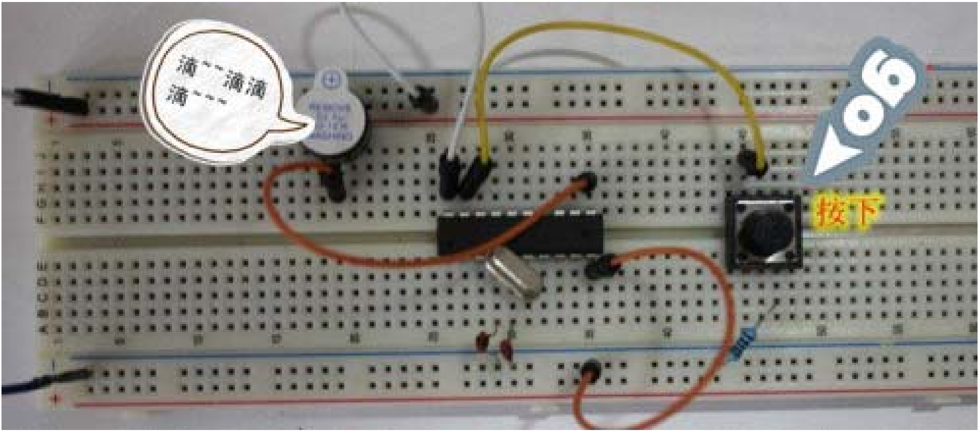


图 1-54 按键控制蜂鸣器实验现象

1.9 动手搭建电路点亮 1602 液晶屏

1.9.1 实验说明

玩点更加刺激的，点亮 1602 液晶屏显示文字。

1.9.2 实验原理图

本次实验的原理图如图 1-55 所示，该部分为 1602 液晶屏电路图，通过我们的单片机去控制 1602 液晶屏显示我们需要的字符数据。

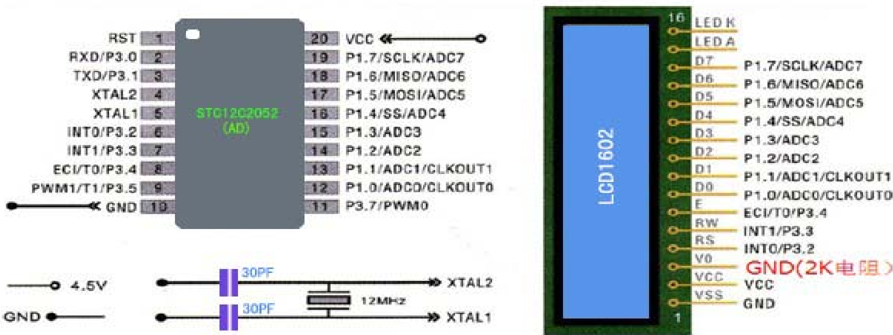


图 1-55 1602 液晶屏实验电路图

1.9.3 器件清单与连接方法

本次实验中，我们所要用到的器件材料如下表 1-14　1602 液晶屏实验器件清单：

表 1-14 1602 液晶屏实验器件清单

品名	型号	数量	参考单价（元）	备注
电池盒	四节七号	1	2	保证输出电压在 4.5~5V
单片机	STC12C2052	1	5	可用 STC12C2052AD 替换
晶振	12MHz	1	0.8	
电容	30pF	2	0.01	陶瓷片电容
液晶屏	1602 字符	1	20	16*2 液晶屏模块
电阻	100Ω1/4W	1	0.01	
面包板	2.54mm 间距	1	10	颜色不限
面包板导线		20	0.01	一捆 10 元

器件所需材料的准备如图 1-56：

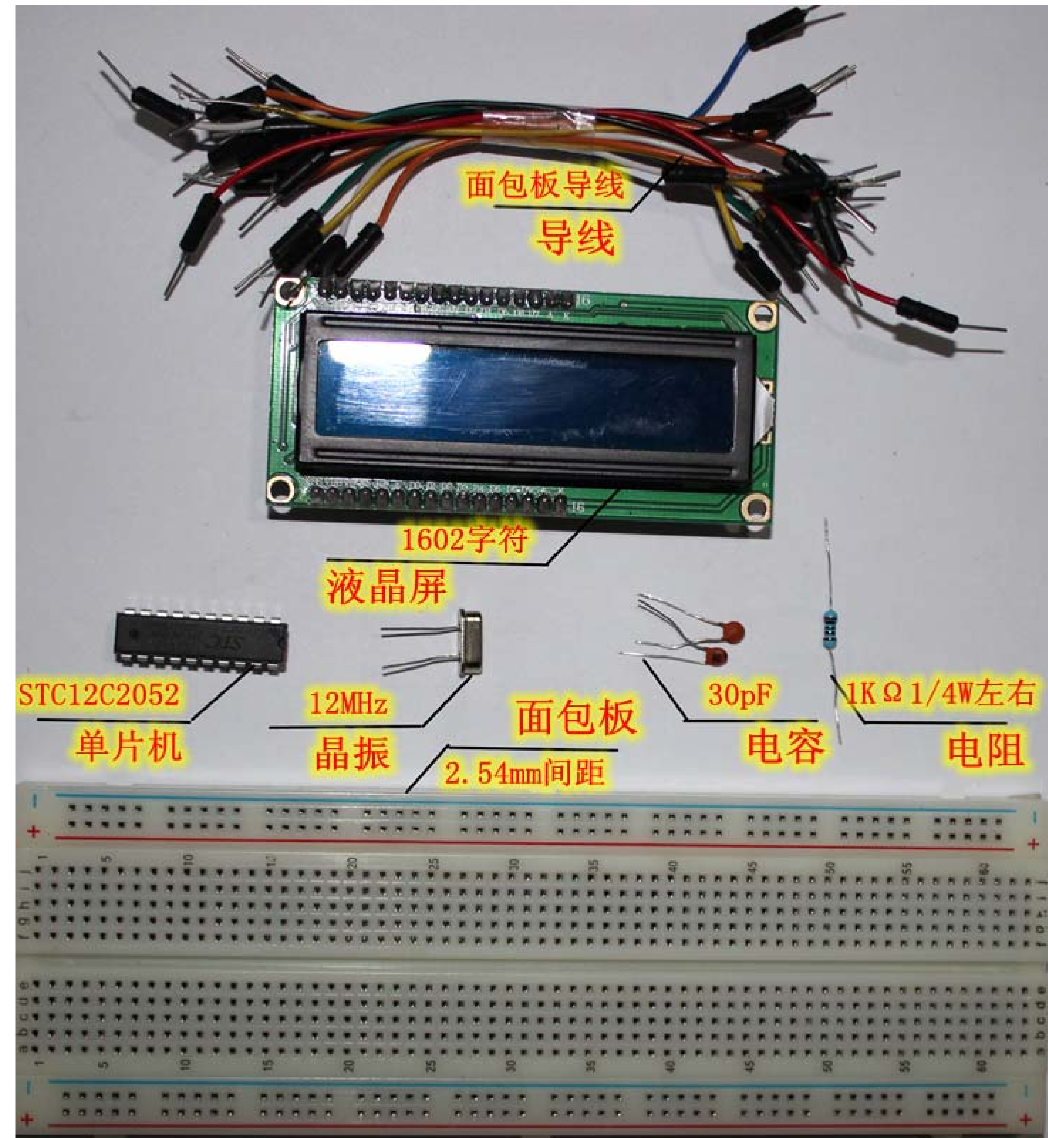


图 1-56　实验所需材料

操作步骤：

1）了解面包板结构，分清各点之间的连接关系

2）查看电路图，点清配套元器件是否满足电路图所需元器件的数量

3）根据原理图在板子上搭建硬件环境，可参考实物图

4）检测搭建电路，可用万用表检测点与点之间的连接

5）烧录相对应的程序到单片机芯片上（配套的程序）

6）查看实验现象（1602 液晶屏显示需要打印的字符）

7）出现问题排查，检测元器件的好坏，对比原理图与所做板子的对比连接

根据原理图把我们的硬件环境搭建起来，如下图 1-57 1602 液晶屏实验硬件环境图：

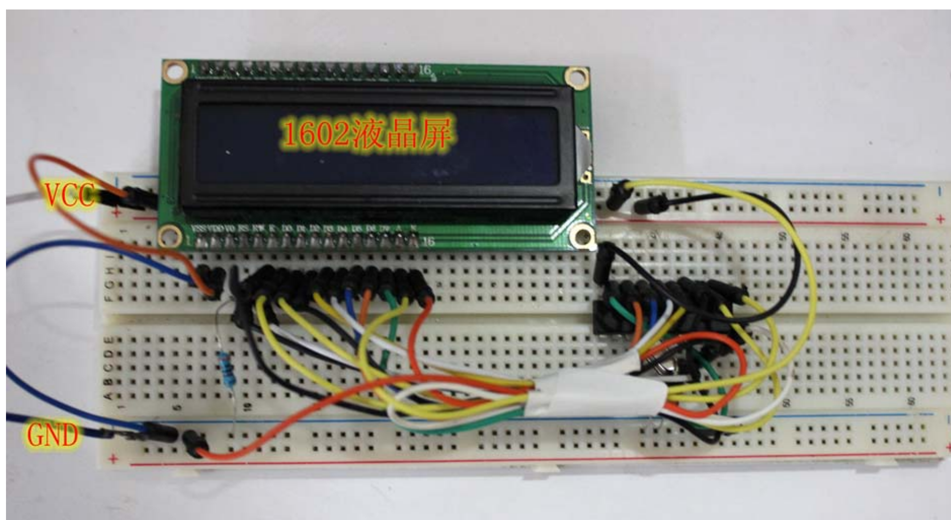


图 1-57 1602 液晶屏实验硬件环境图

1.9.4 程序下载

打开已下载的本书配套资料，选择“步骤 6 51 单片机自己动手搭建最小电路\步骤 6 51 单片机自己动手搭建最小电路\4.DIY 1602 液晶显示(神舟 51+ARM 之 DIY 自己动手搭建电路篇)\DIY1602.hex”文件，使用 STC 官方下载软件进行下载。

1.9.5 代码分析

这次的代码可能有点复杂，但是明白了我们的 1602 液晶屏的时序后，知道什么情况下发送数据到我们的 1602 液晶屏上后就比较好理解了关于 1602 液晶屏的介绍我们的神舟 51+ARM 单片机有单独的一章节对它进行了一个详细的分析了，这里我们只是先看下。

下面的代码是调用的头文件与管脚之间的声明，头文件比我们前面的几个程序中多了一个<intrins.h>，这里我们用的为空操作的指令（_nop_()），所以要调用该头文件。

```
/*****************************************************************************
*  例程：LCD1602 移屏滚动显示字符
*  作者：www.armjishu.com
*  版本：v1.0
*  内容：单片机控制 LCD1602 液晶模块移屏滚动显示 ASCII 字符
```

```
* 现象:通过本例程了解 LCD1602 液晶模块滚屏显示原理,每一行最多储存 40 个字符,
*           液晶屏显示其中的 16 个理解并掌握通过单片机驱动 LCD1602 液晶模块的方法。
*           如果看到 LCD1602 液晶模块的两行滚动显示"www.ARMJISHU.com"等等字符串
*           表明 1602 液晶模块移屏滚动显示正常
*引脚定义如下：1-VSS 2-VDD 3-V0 4-RS 5-R/W 6-E 7-14 DB0-DB7 15-BLA 16-BLK
*************************************************************************/
//包含头文件，一般情况不需要改动，头文件包含特殊功能寄存器的定义
#include<reg52.h>
#include<intrins.h>
//定义端口
sbit RS = P3^2; //RS 为寄存器选择，高电平时选择数据寄存器、低电平时选择指令寄存器。
sbit RW = P3^3; //R/W 为读写信号线，高电平时进行读操作，低电平时进行写操作。
sbit EN = P3^4; //E 端为使能端，当 E 端由高电平跳变成低电平时，液晶模块执行命令。

#define RS_CLR RS=0
#define RS_SET RS=1

#define RW_CLR RW=0
#define RW_SET RW=1

#define EN_CLR EN=0
#define EN_SET EN=1

#define DataPort P1
```

声明完后依然是我们的延时函数

```
/*------------------------------------------------------
 uS 延时函数
 含有输入参数 unsigned char t，无返回值
 unsigned char 是定义无符号字符变量，其值的范围是
 0~255 这里使用晶振 12M，精确延时请使用汇编,大致延时
 长度如下 T=tx2+5 uS
------------------------------------------------------*/
void DelayUs2x(unsigned char t)
{
    while(--t);
}

/*------------------------------------------------------
 mS 延时函数
 含有输入参数 unsigned char t，无返回值
```

```
 unsigned char 是定义无符号字符变量，其值的范围是
 0~255 这里使用晶振 12M，精确延时请使用汇编
------------------------------------------------------*/
void DelayMs(unsigned char t)
{
    while(t--)
    {
        //大致延时 1mS
        DelayUs2x(245);
        DelayUs2x(245);
    }
}
```

下面的判忙函数为我们的判断是否能往 1602 液晶屏传送数据的函数，满足条件才能发送数据。

```
/*------------------------------------------------------------------------------------
                                判忙函数
------------------------------------------------------------------------------------*/
bit LCD_Check_Busy(void)
{
    DataPort= 0xFF;
    //当 RS 为低电平 R/W 为高电平时可以读忙信号
    RS_CLR;
    RW_SET;
    EN_CLR;
    _nop_();
    EN_SET;
    return (bit)(DataPort & 0x80);
}
```

我们往 1602 传送的内容分 2 种，一种是指令，一种是数据，下面的 2 个函数为分别传送指令与数据的函数

```
/*-----------------------------------------------------------------------------------
                              写入命令函数
------------------------------------------------------------------------------------*/
void LCD_Write_Com(unsigned char com)
{
    while(LCD_Check_Busy()); //忙则等待
    //当 RS 和 R/W 共同为低电平时可以写入指令或者显示地址
    RS_CLR;
    RW_CLR;
    EN_SET;
    DataPort= com;
    _nop_();
```

```
        EN_CLR;
}
/*------------------------------------------------------------------------
                            写入数据函数
------------------------------------------------------------------------*/
void LCD_Write_Data(unsigned char Data)
{
        while(LCD_Check_Busy()); //忙则等待
        //当 RS 为高电平 R/W 为低电平时可以写入数据
        RS_SET;
        RW_CLR;
        EN_SET;
        DataPort= Data;
        _nop_();
        EN_CLR;
}
```

清屏函数，清屏的指令为 0x01:

```
/*------------------------------------------------------------------
                                        清屏函数
------------------------------------------------------------------*/
void LCD_Clear(void)
{
        LCD_Write_Com(0x01);
        DelayMs(5);
}
```

写入字符串函数能实现在第一行还是第二行的什么位置显示什么字符数据，y 等于 0 的时候在第一行显示，否则在第二行显示

```
/*------------------------------------------------------------------
                            写入字符串函数
------------------------------------------------------------------*/
void LCD_Write_String(unsigned char x,unsigned char y,unsigned char *s)
{
        if (y == 0)
        {
                LCD_Write_Com(0x80 + x);        //表示第一行
        }
        else
        {
                LCD_Write_Com(0xC0 + x);        //表示第二行
        }

        while (*s)
```

```
	{
		LCD_Write_Data( *s);
		s ++;
	}
}
/*----------------------------------------------------
                         写入字符函数
----------------------------------------------------*/
void LCD_Write_Char(unsigned char x,unsigned char y,unsigned char Data)
{
	if (y == 0)
	{
		LCD_Write_Com(0x80 + x);
	}
	else
	{
		LCD_Write_Com(0xC0 + x);
	}

	LCD_Write_Data( Data);
}
```

1602 液晶屏的初始化函数，这个可以根据我们的 1602 液晶屏工作时序图来操作，只有 1602 液晶屏初始化成功的时候才能正常的工作。

```
/*------------------------------------------------------------------
                         初始化函数
------------------------------------------------------------------*/
void LCD_Init(void)
{
	LCD_Write_Com(0x38);	/*显示模式设置*/
	DelayMs(5);
	LCD_Write_Com(0x38);
	DelayMs(5);
	LCD_Write_Com(0x38);
	DelayMs(5);
	LCD_Write_Com(0x38);
	LCD_Write_Com(0x08);	/*显示关闭*/
	LCD_Write_Com(0x01);	/*显示清屏*/
	LCD_Write_Com(0x06);	/*显示光标移动设置*/
	DelayMs(5);
	LCD_Write_Com(0x0C);	/*显示开及光标设置*/
}
```

介绍完这些函数后，下面我们进入主函数看下，根据前面的介绍，在 1602 进行了一个

初始化与清屏之后，通过我们的写入字符串函数在第一行的起始位置显示"Welcome to SZ-51 armjishu.com 0123456789"，在第二行显示"www.ARMJISHU.com Welcome SZ-51 <ABCDEFG>"，经过一段时间的延时后，我们发送一个指令“0x18” 左平移的指令，使得我们看上去这些字体往左边移动，而发送“0x1C” 是右平移。

```
/*-------------------------------------------------------------
                    主函数
-------------------------------------------------------------*/
void main(void)
{
    LCD_Init(); //初始化
    LCD_Clear();//清屏

    //每一行最多储存 40 个字符，液晶屏显示其中的 16 个
    LCD_Write_String(0,0,"Welcome to SZ-51 armjishu.com 0123456789");
    LCD_Write_String(0,1,"www.ARMJISHU.com Welcome SZ-51 <ABCDEFG>");
    while (1)
    {
        DelayMs(200);
        DelayMs(200);
        LCD_Write_Com(0x18);//左平移画面  0x1C  是右平移
    }
}
```

1.9.6 实验现象

程序下载到芯片上后，给它供电，即可按照程序的设定，在 1602 液晶屏上显示出自己想要的字符如下图 1-62 1602 液晶屏实验现象所示（字符是移动的，所以拍的时候会有点阴影存在）。



图 1-58 1602 液晶屏实验现

第二篇 51 单片机理论深入篇

2.1 学习好单片机的四个步骤

2.1.1 初学者的困难

单片机应用非常广泛，在电的行业和自动控制的应用都非常多。因此许多同学都想学好单片机。有些同学正在学习单片机，有很多同学遇到一些困难，有一些同学单独买了一些单片机书籍，从头到尾一直看，刚开始的时候还能看进去。往往看到中间的时候就看不下去了。即使你把这本书从头到尾都看完了，还是不知道单片机是什么，单片机怎么用。有的同学在网上买了开发板后想自己练，买了一个星期，程序都没同步下载过。身边没有一个专业的人去指导，或者是想通过项目来锻炼自己，没任何机会，针对这种情况，我们这本书就是这样一套手把手的教着大家来学习单片机的教材。

另外，在这里把一些常见问题总结了一下：

问题 1：学会单片机能做什么？职业发展前途怎么样

答：其实这里所说的单片机包含偏硬件方向和偏软件方向两种类别，随着现代化的推进，出现越来越多替代人类工作的新发明和新产品，这些产品的实现都离不开电子产品以及智能软件和互联网，而单片机属于电子控制产品的中心核心的主芯片，例如安防监控，各种智能 IC 卡，汽车电子，摄像机，制造设备，玩具，电子宠物等等都离不开单片机；所以说这是一个替代人类工作的技术，至于说未来这个职业能做什么，那一定是随着时代的发展而发展的。在目前看来，主要体现具体的工作职业名称是，硬件工程师，单片机工程师或者嵌入式工程师等。

问题 2：学习单片机需要什么基础？

答：这个问题也是很多初学者关心的，简单分析，智能的电子产品，没有硬件就好比画饼充饥，所以硬件一定是要有，硬件是由一推电子元器件和 PCB 电路板组成的，这里每个电子元器件又是各种材料制作而成，里面有很多的物理和电的知识，而 PCB 电路板是需要理解电路板的设计知识，懂了这些俗称为硬件工程师；而智能的电子产品的智能部分一定是由软件来实现，因为硬件本身的动作都是由软件来控制的，所以学习这个软件在单片机中一般是使用 C 语言，早期也使用汇编，不过现在已经非常少了，绝大部分都是使用 C 语言来设计软件程序。

问题 3：学习单片机需要用什么软件？

答：这里分两块讨论，一是硬件，硬件设计出 PCB 电路板来，而这些电路板专门工厂生产出来的，工厂收到硬件工程师的 PCB 图按照图的线路进行生产；而这些 PCB 图是一个小文件，可以通过邮箱发送，这个小文件是由各种 PCB 软件设计出来的，比如 PROTEL99，PADS 等公司的软件都是专业设计 PCB 的制图工具，如果要当硬件工程师，就需要学习使用。

另外是软件，最终下载或烧录到芯片里运行的是二进制代码，一般文件格式是 bin 或者 hex 格式结尾，这些文件是用软件编译出来的，什么是编译？编译就好比是翻译，把复杂的程序翻译成二进制的过程就是编译，写程序可以用 C 语言，也可以汇编或者 C++，但可能编译出来可能是相同 HEX 文件，只要是功能相同就基本一致。那么这里就需要使用一个软件来实现写代码以及编译工作，目前使用比较多的是三种，KEIL，IAR 以及 ADS 软件。

问题 3：学习单片机有什么好的教材推荐？

答：教程非常多，在这里举个小例子，假如这些教程都是要完成描述一条路如何从开始

走到终点的整个过程，那么有一部分教程主要侧重描述走这段路的心得体会；有些教程主要描述从开始到开始的前半部分；有些教程描述中间到后半部分；其实内容都有侧重，所以很多初学者不会挑选，也很茫然，很难走完整条路，因为很难买到一本很全面的书籍，并且一本书籍也无法把整个过程描述那么清楚，毕竟受篇幅限制；我们本书基本上是把整条路的一些关键知识点做了穿透性的打通，目的是让初学者具备基本的能力之后自己可以独立解决一些更难的问题，主要侧重打基础为主，至于说需要一些其他的资料补充，请初学者学习完之后根据自身的情况再去做进一步的选择和补充。

2.1.2 学习单片机的四个步骤

学习单片机经过多年总结以及与许多初学者沟通，归纳下来有四个步骤值得关注：

第一步，一步一个脚印跟着走。

在这个过程当中呢，就是大家跟着用户手册和视频教程一步一步来操作，严格根据教程的说明，如何连线，如何进行软件操作、硬件操作等；刚开始学习，只要知道它是什么就可以了，如同小孩子学说话，大人叫他叫爸爸、妈妈，他其实根本不知道这两个词语是什么意思，但是，你要不停的教他，他就不断的模仿，模仿一段时间后他就会叫了，再过一段时间，他就慢慢理解了，因此呢，刚开始大家只要记住就可以了，不需要知道为什么，这个是第一个过程。这样懵懂的坚持一段时间之后，会自然而然产生一些体会和总结来的，说得更直白一些就是通过动手来产生理解和感觉，这个理解逐渐加深的过程需要一点时间。

第二步，照葫芦画瓢

第二个过程，我认为是最重要的一个过程，还是刚才的例子，刚开始他不知道爸爸、妈妈是谁，但后来他知道后就会主动的叫了，这个是一个什么过程呢？这里就是同学们看了用户手册和视频教程学会之后，就可以把教程给关掉，关掉然后自己亲自去把握上面的实验，独立的去完成，这个过程我认为是特别特别的重要，强力的推荐初学者把这个第二个过程从头到尾把所有的实验都实现一遍，操作一遍。这里有个误区就是很多看似会了，自己一动手，肯定傻眼，就是一个简单的十多行程序，我遇到一个同学，自己写的时候，大小写他不区分，很容易就忽略掉，就是一些细节问题，他听的时候没注意，他自己写的时候就写不出来，我希望所有的同学在求学的过程中，多去做这种实践，自己提前去实践，把这些问题遇到，以后真正去做项目的时候，这些问题就会去注意，因此呢，我认为第二步，特别是初学者最重要的一步，把我所有的实验，你能够做到，关掉用户和视频，然后独立把这个程序做好，这个是第二步。

第三步，站在巨人的肩膀上看世界。

这个过程是一个什么样的过程呢？因为像我们平时在项目开发的过程中，就常用到的一种方式，我们在设计一个新东西的时候，首先要查找国内外所有的公司，所有的研发机构，他们做的同类产品是怎么样的，因为如果你自己亲自去设计，亲自把这个程序写出来，亲自把电路设计出来，难免考虑不周全，肯定会有很多漏洞，因此你把别人的这个设计、这个理念，这个程序拿过来，首先研究一下，看看别人怎么做的，然后把别人几个稍微综合一下，然后研究明白了，修改加以改进，然后能有突破是最好的，突破就是你这个程序，这个项目，这个就是在日常生活中用到的一个方法。

第四步，理论联系实践。

刚开始看书的话，没任何目的性，看书你不知道看什么，不知道哪些东西应该要记住，哪些东西可以一扫而过，你不可能把整本书或者几本书都背下来，那是不可能的，因此呢，第四步就真正的是理论实践结合，我们在实践操作的过程中，什么地方有不明白的地方，然后回头去查书，书上的知识点在哪个地方，把这块仔细研究明白，这个就是一个理论实践结

合，但你研究明白的时候，不明白的再回来看书，这个一个缓速的过程，就可以把自己的一个能力提升起来了，这个是第四步，理论实践结合。我相信很多同学，特别是初学者根据这四步做起来，你也可以经常回头看下这四步，自己当前处在一个什么阶段，把这四步全部落下来，基本上可以上手了，也就是说你进了公司之后啊，有高手带着你，可以跟着别人做开发了，这个是学习单片机的四步曲，根据学习单片机的这四个步骤总结起来，一个宗旨，在实践中成长，这个是我要求大家多写程序，多调电路，这个是一个最重要的过程，我比较反对的是只用软件仿真，不亲自去焊板子的那种做法，因为我们懂了这个知识点，很多经验教学是在实践中练出来的，就如同老铁匠带着徒弟打铁，他都是在打铁的过程中告诉徒弟什么时候扔锤子的，什么时候锤子抡到什么程度，在实践中慢慢磨练，才能慢慢成长起来，如果只看理论的话是不行的。

第五步，项目实战。

进入这个阶段的时候，一定已经有了成熟的知识体系架构，当然任何理论上的知识架构都要经得住实战的考验，所以通过做实际的项目来发现自己的不足，来补充欠缺的知识，不断完善巩固自己的知识框架；修正自己的学习方法，这条深入来说就是前面入门过程是一套学习方法，项目实战的时候是另外一套学习方法，实战讲究把单点的知识点挖深挖透，或者遇到新的问题一种解决问题的能力，为了尽快完成项目，这就需要形成自己的一套解决问题的方法和思路，最后把新获得知识再存放回自己的知识架构中，进而水平能力都得到大大的提升。更多这方面的知识，本书最后的章节会有更进一步的剖析。

2.1.2 学习单片机的准备工作

学习好 51 单片机的到底要做好哪些准备呢？主要是三个准备：

- 第一是信心与恒心。学习单片机你要相信自己，就像是一个窗户纸，在外面看非常神秘、非常复杂、非常难，只要有信心用来去捅破这层窗户纸，其实是非常简单的，我们的单片机任何一个程序、一个电路我们都能找出他的原因，第二个是一个恒心，有些同学在调程序的时候，可能二三天都调不出来，这个时候不要气馁，这个是很正常的，只要你把前面的困难过了，后面调程序就很快了，大家要坚持下来。因此信心和恒心是必不可少的。
- 第二是有一本手把手操作的实验教材和一本 C 语言教材。手把手的实验教材可以帮助你从零基础入门开始到精通的一个过程，它会指引着你前行；另外一本是 C 语言教材，这个 C 语言教材不是给你从头到尾一直看的，当遇到哪些地方不明白的时候，查找目录，找到解决问题的地方，书是用来查的，我们现在不是考试的过程，是能查阅资料的，在你以后的项目中也肯定需要去查阅大量的资料。
- 第三部分，实践操作所需要的环境，这里包括电脑，实验开发板，以及其他的工具。这个虽然不是学习单片机必须的，但是要学好单片机，我认为是不可缺少的，这个东西就如同军队去军事演习，开发板就是一个平台，就是给你一个锻炼的平台，你可以通过开发板去锻炼自己，通过开发板的学习，你把绝大部分的电子器件都了解一下，跟别人去开发项目的时候都能马上明白，在这里我们推荐一下我们的神舟 51+ARM 开发板，当然你也可以自己做，前提是你要明白原理等内容。

2.2 单片机芯片入门理解

2.2.1 处理器如何控制一个智能产品

处理器如何控制一个智能产品呢？我们举个例子，比如人就是一个智能产品，这个智能产品主要是靠人的大脑来控制的，人的大脑就好比是一颗 CPU 处理器，人脑来负责控制整个人的行动，四肢，思想等，包括说话，笑、哭等情态的动作。

处理器与人的大脑是一个原理，处理器通过芯片的管脚来输入输出达到控制智能产品的目的；比如无论我们大脑如何复杂，我们控制自己的四肢的时候，都是一步一步来的，比如伸手，比如弯腰，输出都是比较简单的动作；比如愚公移山，也是愚公他自己一块石头一块石头经过许多年时间才移走的，所以处理器也是同样的道理，它虽然内部处理的信息非常复杂，但它周围的芯片管脚也是一个信号一个信号的输出来或者输入进去，来达到控制所谓复杂的智能产品的。

所以尽管非常复杂的产品也好，或者复杂的人也好，其实对改变外界改变自然来说，都是一系列最简单的动作组合起来的，因此不要认为处理器内部处理的事务特别复杂，虽然我承认确实是复杂的，但我在这里想表达的意思是它是有规律，有顺序的一个一个执行的。

一系列简单的动作，一系列简单思维，一系列有调理的流程进行有条理化的组合就变成了处理器里的程序，程序就是用来干这个事情的，程序一般都是由工程师开发的，比如有控制电冰箱的程序，有控制电梯的程序，有控制遥控飞机的程序，有控制电饭煲的程序等等，程序通过运行来监控处理器外部被控制器件的状态和反馈，把这些反馈输入到正在运行的程序中，采取应有的措施，而这些措施都是靠芯片的管脚进行输入输出来表达的。

2.2.2 处理器芯片管脚的理解（不是输入就是输出）

如果要说做单片机很难吗？任何芯片包括 51 或者 ARM 其实都不难，最基本的原理用 3 句话就可以讲明白：

第 1 句话：一个芯片管脚要么是输入，要么是输出。

所有的程序，用单片机控制的产品，以及外设，无非就是控制芯片的各个管脚输入或者输出两个状态；例如，芯片发送数据就是输出；芯片驱动一个产品，也是输出；芯片接收数据就是输入；单片机对一个存储芯片写输入，可以理解为单片机与存储芯片连接的管脚输出状态，输出数据到存储芯片的管脚上，而存储芯片此时它的芯片对应管脚被配置成输入，将数据写入到芯片内部。

所以说，芯片管脚不是输入，就是输出，当然，如果你不使用这个管脚，也可以将它配置成某一种中间状态，免得干扰了外界，影响了 PCB 板上的其他元器件状态。

第 2 句话：芯片管脚不是高电平，就是低电平。

芯片管脚不是高电平就是低电平两种状态，当然也有第三种，既不高电平也不是低电平的状态，这样的管脚状态表示没有任何内容和数据；无论管脚是输入还是输出，它的目的都是传输数据、传输信息，所以管脚的高电平我们将它表示为“1”，低电平表示“0”，通过 0 和 1 这样的数据来传输它想传输的内容，这个就是所谓的二进制。

例如：假如复位芯片管脚是低电平进行复位，我们将该管脚一直拉高为高电平“1”的时候，芯片可以正常工作，如果将管脚拉低至低电平“0”的时候，芯片通过检测这个管脚状态为低电平，芯片内部就会自动进行复位；我们通过控制这个管脚拉高和拉低，从而就可

以达到控制芯片的工作；其他的管脚也是同样的道理。

第 3 句话：传输协议。

什么是传输协议，比如与串口芯片通信，那么就要是串口协议的；如果是 I2C 协议的 EERPOM，那么就是 I2C 协议；还有其他一些比如 485 协议，CAN 协议，USB 协议，SD 卡的 SDIO 协议…….等等数不胜数。

而这些协议，无非就是按照预先规定的表达方式进行通信，比如举个例子，我约定先连续发 4 个 1,，然后再发 4 个 0，就表示芯片 A 要开始发数据给芯片 B 了，即芯片 A 通过它的芯片管脚发‘11110000’给到芯片 B 的时候，那么芯片 B 就知道芯片 A 要给它真正的数据，它就要做好准备工作，准备好之后，芯片 B 就会给芯片 A 一个回应，当芯片 A 收到芯片 B 的回应，就正式开始发数据。

这样通信双方之间的协商规定，就构成了协议，经过这么多年，就形成了我们所常见到的串口协议，CAN 协议，USB 协议（像 USB 协议又分为 USB1.0 协议，USB2.0 协议，USB3.0 协议，版本越高，速度就越快，协议进行优化后，通信效率也变高了）。

所以总结下来，一个芯片最简单的外设莫过于 I/O 口的高低电平控制，只要掌握了 I/O 管脚的输入和输出，高电平与低电平控制，再理解传输的协议，就基本算是掌握了单片机的本质了，在这里下面将详细讲解一下如何用一个 I/O 口去控制一个 LED 灯的亮灭，由最基础的例程入门吧。

2.2.3 处理器怎么认识下载进去的程序代码的？

这个问题很搞笑也很实在，处理器怎么认识下载到处理器内部的程序的？并且它还能按照程序的要求一步一步的执行，进行相应的动作。

为了说明这个问题，我们举个沟通之间的例子。假如你遇到一个老乡，我们可以彼此讲家乡话进行聊天；假如这个时候来了一个其他城市的人，他听不懂你的家乡话，那没有问题，我们还有普通话进行沟通，当然前提是你和他都会讲普通话；这个时候不巧，刚好来了个国外的人，比如是英国人或者美国人，这个时候如果你会讲英语的话，那么你们就可以畅快的聊天和沟通。

上面这个例子中的主人公面对 3 个不同的人换了 3 种语言，这能说明什么问题呢？为什么要换 3 种不同的语言呢？答案就是为了让另外一个人听得懂，说对了，就是让另外一个人听懂我们的语言。

那么处理器怎么听懂我们讲的话呢？哈哈，它当然是听不懂的，你必须要给处理器发明一种新的语言，让处理器能听得懂的，并且我们程序员也要听得懂的才行；比如在程序里随便写一个 3 位数比如 001，如果处理器见到 001 这 3 个数，它就把自己的第一个管脚输出高电平，这个动作是事先在处理器中约定好了的，由设计和生产这个处理器的公司负责做好；假如 002 表示第 2 个管脚输出高电平，003 表示第 3 个管脚输出高电平等….这里看到没，处理器竟然可以被控制了，我们只需要输入它能看懂的对应代码就能灵活的控制处理器了。我们暂定为称这些命令为指令吧，一个指令就可以命令处理器干一件事情。

那么如果把更多的指令建立起来，那不是可以让处理器干更多的事情，对！这样就可以了，其实程序代码就是一系列指令组合起来的，处理器看不懂几万行代码程序，但它懂得如何一行一行的执行程序代码。

这些指令组合起来的集合就被官方术语称之为指令系统，每个芯片都有自己出厂的一套指令系统，不同芯片的指令系统可能有不同，那么它怎么识别到我们写的程序呢？

比如我们写的 C 语言程序，其他语言的程序都是同样的道理和原理，C 语言编写代码后，在编译前有个设置，设置目标版的处理器芯片型号，此时，其实就选择了一套针对这个

芯片的指令系统，那么在编译的时候，就会把我们写的 C 语言代码程序翻译成这个指令系统中的一系列指令，编译的工作就是按照这个指令系统进行翻译，最后变成 0101 的二进制文件，CPU 的指令都是由 0101 的二进制组合起来的。

2.2.4 为什么采用二进制，而不使用三进制，四进制

二进制是计算技术中广泛采用的一种数制。二进制数据是用 0 和 1 两个数码来表示的数。它的基数为 2，进位规则是“逢二进一”，借位规则是“借一当二”，由 18 世纪德国数理哲学大师莱布尼兹发现。当前的计算机系统使用的基本上是二进制系统。

计算机是怎么使用二进制的呢？二进制在计算机中都是 01010101 的数，不是‘0’就是‘1’；上一节讲了指令系统，计算机内部被约定每 8 个单个 0 或者 1 的数字就为一个字节，这个 8 个数可以灵活的组成，变成各种各样的指令，比如 8 个都是 0 表示关机，8 个都是 1 表示开机（这里只是举例子），那么这样的方式就可以让计算机通过二进制读懂人类给它发的指令，建立起一个沟通的渠道。

那么这里有一个新的问题，三进制可以执行吗？答案是可以，我们接下分析一下，二进制是高电平称之为 1，低电平称之为 0；那三进制，我们假如有高电平，地（即零电平），低电平三种状态，那么可以用 1 表示高电平，0 表示地，-1 表示低电平，其实这样也是可以的。在理论上是可以行得通的。四进制也是同样的道理，也可以的。

那为什么要用二进制，而不是使用其他的进制呢？这里主要是因为二进制有个优势，这里对比二进制和三进制，假如二进制有个高电平数据发过来，三进制有个低电平数据发过来，在传输过程中，突然受了一个外界电磁波的干扰，二进制发过来的高电平电平经过干扰后干扰前更高了，三进制的低电平经过干扰后，本来是负电平的，可能被干扰成零电平了，就是地，可以看到，三进制的数据传输到目标端的时候，数据出现了错误。而二进制却不会。

二进制的优势在于符合大自然的规律，不是白天，就是黑夜；不是白就是黑；不是阴极就是阳极；不是同意就是反对；不是高电平，就是低电平；很清晰很容易辨别，使得抗干扰性比较好，想想看，如果是三进制，高中低电平三种状态，需要认为去划分，划分的间隙不够的话，随便干扰一下就跳到另外的段了。二进制如果是低电平，比如是 0，随便怎么干扰接地就是接地，没有什么反映，如果是高电平，随便怎么干扰，它还是有电平状态的，有电平状态就是 1，这样就很好去判断了。

2.2.5 处理器硬件上如何实现存储二进制数的？

触发器是一种电路单元，它有两个稳定的工作状态，一个为 1，一个为 0；这个工作状态是触发器这个电路单元的输出端的状态，在没有外接信号作用时，触发器维持原来的稳定状态，只有在一定外接信号作用下，触发器可以从一个稳定的工作状态翻转到另一个稳定状态。

触发器是一个可以记忆二进制信号 0，1 的存储单元，这就是触发器的记忆作用，触发器的工作状态不仅和输入端有关，而且还和过去的工作状态有关。

一个触发器只能寄存一位二进制数 ，要储存多位，就要用多个触发器；比如记录 8 位二进制数，就要 8 个触发器，如果一个寄存器是 8 位的，那么这个寄存器本身就可以由 8 个触发器组成。触发器可以保持状态，就可以实现寄存器被设置后，也一样是保存状态。

所以处理器硬件上，包括寄存器，都是由大量的触发器单元电路组成的，如果放在显微镜下，那是多么壮观的一个场面。

2.2.6 单片机芯片的选型

单片机的种类非常多，各种型号看得眼花缭乱，下面通过讲一下单片机内部资源的三大指标，来说明如何通过单片机的种类来选择单片机：

第一个指标是 FlASH，Flash 是程序存储空间，也就是说把我们 C 语言开发好的程序最终下载到单片机的 Flash 区域内，控制单片机的工作。这是 Flash，主要是存储我们程序的。

第二个呢，第二个指标，RAM，RAM 是内存，内存呢，就是主要存储我们定义的一些变量。还有一些中间阶段的结果，存储在 RAM 当中。这个 Flash 和 RAM 对比来说，这个 Flash 类似于我们电脑的硬盘，这个 RAM 相当于我们电脑的内存。为什么选择 Flash 当程序存储空间，这个 Flash 它有一个特点，就是掉电后程序不会丢失，也就是说我们程序写到单片机内部之后，这个系统运行，运行突然停电了，下次我们再上电的时候呢，这个程序还在单片机内部保存着，Flash 就有这样一个特点。因此它就是用来存储程序，而 RAM 呢，它的第一个特点就是无限次擦写，也就是说我们很多同学在配电脑的时候，他们的内存厂家是终身保修的，所以内存理论上是可以无限被擦写，如果你的内存坏了的话，你的电脑其他的设备估计也都已经坏了。这个是内存的第一个特点，第二个，也是最重要的一个，它的存储读取特别快，就是说我们在中间计算的时候，我们要快速的存储数据，快速的读取数据，这样的话，单片机的工作效率更高一点，因此呢，我们使用 RAM 来存储中间变量，用 RAM 刚才也说了，它是能被无限次擦除的，Flash 呢？它的擦写速度慢一点，第二个呢，它是有限制次数的，FlASH 大约 10 万次。大家也不用担心我们的单片机下载程序会把 Flash 下载坏了，因为 10 万次的话，假设每天下载同一个单片机，每天下载 300 次的话，我们还可以使用 1 年左右，因此大家要大胆的去实验，不用担心用坏了，

第三个指标呢，是 SFR，特殊功能寄存器，这个是单片机非常重要的一个指标，就是说单片机在出厂的时候，他的内部剁好了一个一个的小模块，这些小模块将来我们编程的时候要去控制这些小模块，完成整个系统的工作，这个 SFR 现在这样的解释可能会使得大家有些听不明白，不过不用担心，等到后面我们慢慢的给大家再介绍，现在先把这个 SFR 特殊功能寄存器先记住。以后慢慢的去学它，用它的时候，就把这个东西理解了。

在神舟 51 单片机开发板上，选了两款单片机芯片用来学习，一款是 STC89C52，另一款是 STC90C516RD，目标是通过学习这两款单片机达到掌握市面上的 8 位单片机的原理，玩精通这两款单片机之后就不用再去学习其他的单片机了，因为原理都是相同的，因为我们学完这两款单片机之后，其他的八位单片机就要拿到手上来看一下，就会如何编程，如何写程序，我们要达到这样一个目标，因为单片机是千千万万的，很多很多种类，很多很多型号，那么如果你每个型号都去学一下的话，根本不可能的，因此我们的目标是通过这两款单片机学会所有的单片机。第一款单片机，STC89C52，这个单片机是国内的一个厂商生产的。它的特点呢，就是价格比较便宜，做一些小东西特别适合，尤其是对大家 DIY 开始学习的时候，这款单片机是非常合适的，因为价格比较便宜，而且下载程序比较方便，这个是它的特点，它是 8K 的 Flash，Flash 就是程序存储空间，512 字节的 RAM，大家要记得 1K 等于 1024 个字节，这个 512 字节是它的 RAM 的大小，32 个 I/O 空、3 个定时器，这些大家先看一下，以后用到的时候我们再慢慢了解。

第二款单片机是 STC90C516RD，它是 STC 推出的新一代超强抗干扰/高速/低功耗的单片机，指令代码完全兼容传统 8051 单片机，但 ROM 和 RAM 分别为 64K 和 1280 字节，都要超过 STC89C52 许多，方便运行更大的程序代码，例如有的彩屏显示图片等程序本身所占空间就比较大，STC89C52 里的 ROM 空间不能满足这个空间，根本无法下载进去，所以就要使用这种大一点 ROM 空间的 STC90C516RD 这个型号的芯片，并且 RAM 也大一些，运

行的速度也会快很多。

2.3 51 单片机资料阅读方法

2.3.1 如何阅读 51 单片机的芯片手册

打开我们的光盘资料中的芯片参考手册文件夹，找到我们的《STC89C51、52 RC-RD 使用手册.pdf》文档，如下图 2-1 所示：



图 2-1 51 单片机芯片手册文档

将其打开后可看到如下图 2-2 与图 2-3 的介绍：


STC89C51RC / RD+ 系列单片机器件手册
---加密性强
---低功耗,超低价
--- 高速，高可靠
--- 强抗静电，强抗干扰

STC89C51RC,　STC89LE51RC
STC89C52RC,　STC89LE52RC
STC89C53RC,　STC89LE53RC
STC89C54RD+,　STC89LE54RD+
STC89C58RD+,　STC89LE58RD+

图 2-2 单片机文档部分介绍图


STC89C54RD+,　STC89LE54RD+
STC89C58RD+,　STC89LE58RD+
STC89C516RD+, STC89LE516RD+
STC89LE516AD, STC89LE58AD
STC89LE54AD,　STC89LE52AD
STC12C5404AD, STC12C5404
STC12C2052AD, STC12C2052

推荐优先选择代表宏晶最新水平的 STC12C54xx 系列单片机

宏晶科技
www.MCU-Memory.com
Update date: 2007-11-17

图 2-3 单片机文档部分介绍图

这个手册是宏晶科技制作的，可以看下版本，是 2007-11-17 日的文档

接下来查看目录，该文档手册写得有点乱，但是所有的硬件相关内容确实都可以从这个文档中找到，包括寄存器的地址，最小系统设计，参考电路，封装大小，特性等，大家需要

什么资料，就去找什么资料，这样会更好更方便一点。

文档目录如下图 2-4 所示：

5 / 273 125% 查找

目录

图 2-4 51 单片机芯片手册目录

如果资料觉得不够，更多资料可以上网站http://mcu-memory.com去获取，下图 2-5 为网站的部分内容。

图 2-5 宏晶科技网站

这里的资料都厂商公开的，资料非常多，非常丰富，用户可以根据自己的需要下载需要的手册和文档。

2.3.2 51 单片机的管脚是如何被控制的

这是在芯片手册里截取出来的图，比如像 SCON 这个寄存器它的地址是 98h，那么可以从 reg52.h 头文件中找到 sfr SCON =0x98;，用一个 SCON 的符号与 0x98 这个地址挂钩，在 51 单片机的源代码中直接操作 SCON，就等于是操作 0x98 这个地址，再对应一下手册，0x98 就是 98h 这个地址，也就是单片机内部真正的 SCON 寄存器的地址，看下表 2-1：

表 2-1 STC89C51RC/RD+ 系列 8051 单片机 串行口特殊功能寄存器 Serial I/O Port SRFs

Mnemonic	Add	Name	7	6	5	4	3	2	1	0	Reset Value
SCON	98h	Serial Control	SM0/FE	SM1	SM2	REN	TB8	RB8	TI	RI	0000,0000

SBUF	99h	Serial Data Buffer									xxxx,xxxx
SADEN	B9h	Slave Address Mask									0000,0000
SADDR	A9h	Slave Address		-	-	-					0000,0000

所以说，“98h，99h，B9h，A9h”等都是单片机的内部地址，而“Reset Value”这一列都是该寄存器复位后的默认值，比如 98h 的复位后默认值是 0000 0000。

还有更多的关联和挂钩的，截取 reg52.h 文件中的部分代码：

```
/*--------------------------------------------------------------------------
REG52.H
Header file for generic 80C52 and 80C32 microcontroller.
Copyright (c) 1988-2002 Keil Elektronik GmbH and Keil Software, Inc.
All rights reserved.
--------------------------------------------------------------------------*/
#ifndef __REG52_H__
#define __REG52_H__
/*  BYTE Registers  */
sfr P0    = 0x80;
sfr P1    = 0x90;
sfr P2    = 0xA0;
sfr P3    = 0xB0;
sfr PSW   = 0xD0;
sfr ACC   = 0xE0;
sfr B     = 0xF0;
sfr SP    = 0x81;
sfr DPL   = 0x82;
sfr DPH   = 0x83;
sfr PCON  = 0x87;
sfr TCON  = 0x88;
sfr TMOD  = 0x89;
sfr TL0   = 0x8A;
sfr TL1   = 0x8B;
sfr TH0   = 0x8C;
sfr TH1   = 0x8D;
sfr IE    = 0xA8;
sfr IP    = 0xB8;
sfr SCON  = 0x98;
sfr SBUF  = 0x99;
/*  8052 Extensions  */
sfr T2CON  = 0xC8;
sfr RCAP2L = 0xCA;
sfr RCAP2H = 0xCB;
```

```
sfr TL2    = 0xCC;
sfr TH2    = 0xCD;
/*  BIT Registers  */
/*  PSW  */
sbit CY    = PSW^7;
sbit AC    = PSW^6;
sbit F0    = PSW^5;
sbit RS1   = PSW^4;
sbit RS0   = PSW^3;
sbit OV    = PSW^2;
sbit P     = PSW^0; //8052 only
/*  TCON  */
sbit TF1   = TCON^7;
sbit TR1   = TCON^6;
sbit TF0   = TCON^5;
sbit TR0   = TCON^4;
sbit IE1   = TCON^3;
sbit IT1   = TCON^2;
sbit IE0   = TCON^1;
sbit IT0   = TCON^0;
/*  IE  */
sbit EA    = IE^7;
sbit ET2   = IE^5; //8052 only
sbit ES    = IE^4;
sbit ET1   = IE^3;
sbit EX1   = IE^2;
sbit ET0   = IE^1;
sbit EX0   = IE^0;
/*  IP  */
sbit PT2   = IP^5;
sbit PS    = IP^4;
sbit PT1   = IP^3;
sbit PX1   = IP^2;
sbit PT0   = IP^1;
sbit PX0   = IP^0;
/*  P3  */
sbit RD    = P3^7;
sbit WR    = P3^6;
sbit T1    = P3^5;
sbit T0    = P3^4;
sbit INT1  = P3^3;
sbit INT0  = P3^2;
sbit TXD   = P3^1;
sbit RXD   = P3^0;
```

```
/*   SCON   */
sbit SM0      = SCON^7;
sbit SM1      = SCON^6;
sbit SM2      = SCON^5;
sbit REN      = SCON^4;
sbit TB8      = SCON^3;
sbit RB8      = SCON^2;
sbit TI       = SCON^1;
sbit RI       = SCON^0;
/*   P1   */
sbit T2EX   = P1^1; // 8052 only
sbit T2     = P1^0; // 8052 only
/*   T2CON   */
sbit TF2      = T2CON^7;
sbit EXF2     = T2CON^6;
sbit RCLK     = T2CON^5;
sbit TCLK     = T2CON^4;
sbit EXEN2    = T2CON^3;
sbit TR2      = T2CON^2;
sbit C_T2     = T2CON^1;
sbit CP_RL2 = T2CON^0;
#endif
```

可以看到代码中对 P0，P1,P2,P3 都有定义，例如 sfr P0=0x80;可以与下表 2-2 的对应上：

表 2-2 8051 单片机特殊功能寄存器

Mnemonic	add	Name	7	6	5	4	3	2	1	0	Reset Value
P0	80h	**8-bit Port0**	**P0.7**	**P0.6**	**P0.5**	**P0.4**	**P0.3**	**P0.2**	**P0.1**	**P0.0**	**1111,1111**
P1	90h	8-bit Port1	P1.7	P1.6	P1.5	P1.4	P1.3	P1.2	P1.1	P1.0	1111,1111
P2	A0h	8-bit Port2	P2.7	P2.6	P2.5	P2.4	P2.3	P2.2	P2.1	P2.0	1111,1111
P3	B0h	8-bit Port3	P3.7	P3.6	P3.5	P3.4	P3.3	P3.2	P3.1	P3.0	1111,1111
P4	E0h	8-bit Port4	-	-	-	-	P4.3	P4.2	P4.1	P4.0	xxxx,1111

那么如果要访问 P0.0 和 P0.1 这两个管脚该怎么处理呢？只需要使用 sbit 关键词和符号ˆ就可以，例如：

```
Sbit  aa = P0ˆ0;
Sbit  bb = P0ˆ1;
```

就可以使得 aa 代表 P0.0 管脚，而 bb 代表 P0.1 管脚，并且可以看到通过这个 sbit 的定义可以访问到 80h 这个寄存器 P0 里的任何一位，如下表 2-3 所示：

表 2-3 访问 P0 口寄存器

Mnemonic	add	Name	7	6	5	4	3	2	1	0	Reset Value
P0	80h	**8-bit Port0**	**P0.7**	**P0.6**	**P0.5**	**P0.4**	**P0.3**	**P0.2**	**P0.1**	**P0.0**	**1111,1111**

所以 51 单片机就是这样，可以用代码控制所有的寄存器。

2.4 从零开始搭建 51 编程环境

2.4.1 例程环境搭建

1. 点击桌面UVision2图标，如图2-6，启动软件。如果是第一次使用的话会打开一个自带的工程文件，我们可以通过工具栏Project->Close Project选项把它关掉如图2-7。



图 2-6 UVision2 图标

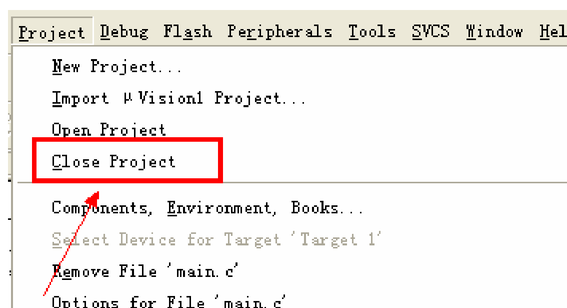


图 2-7 关掉工程操作

2. 在工具栏Project->New μVision Project...新建我们的工程文件如图2-8，我们将新建的工程文件保存在桌面的“神舟51+ARM之STM32F103C8T开发板模板工程”（先在电脑桌面上新建一个“神舟51+ARM之STM32F103C8T开发板模板工程”文件夹，在该文件夹里面新建一个Project文件夹），文件名取为：STM32-DEMO（英文DEMO的意思是例子），名字可以随取，如图2-9，点击保存。

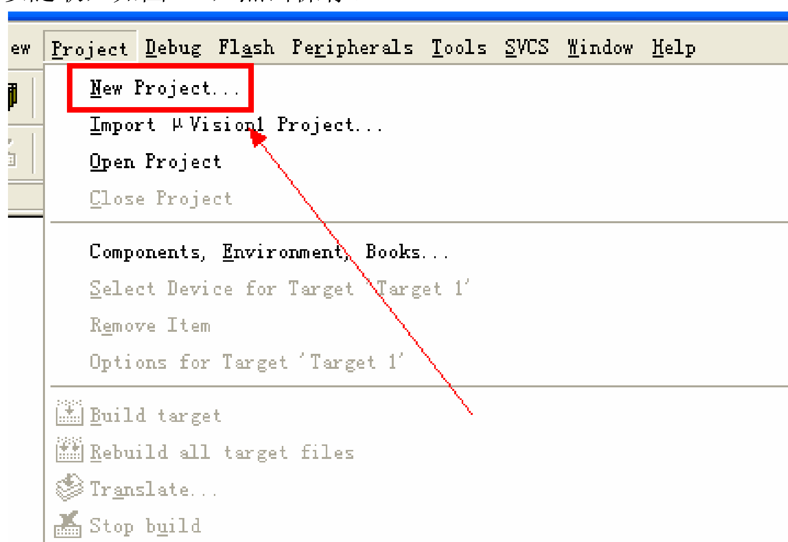


图 2-8 新建工程操作（1）

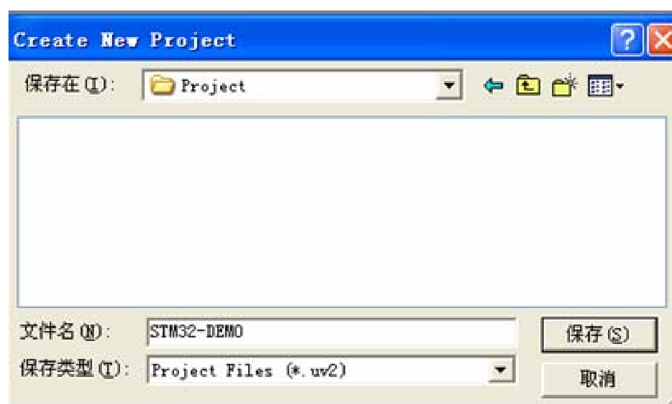


图 2-9 新建工程操作（2）选择保存位置

3. 接下来的窗口是让我们选择公司跟芯片的型号，因为我们用的芯片是 Atmel 公司的 AT89S52，有 8K Flash。按如下选择即可如图 2-10。

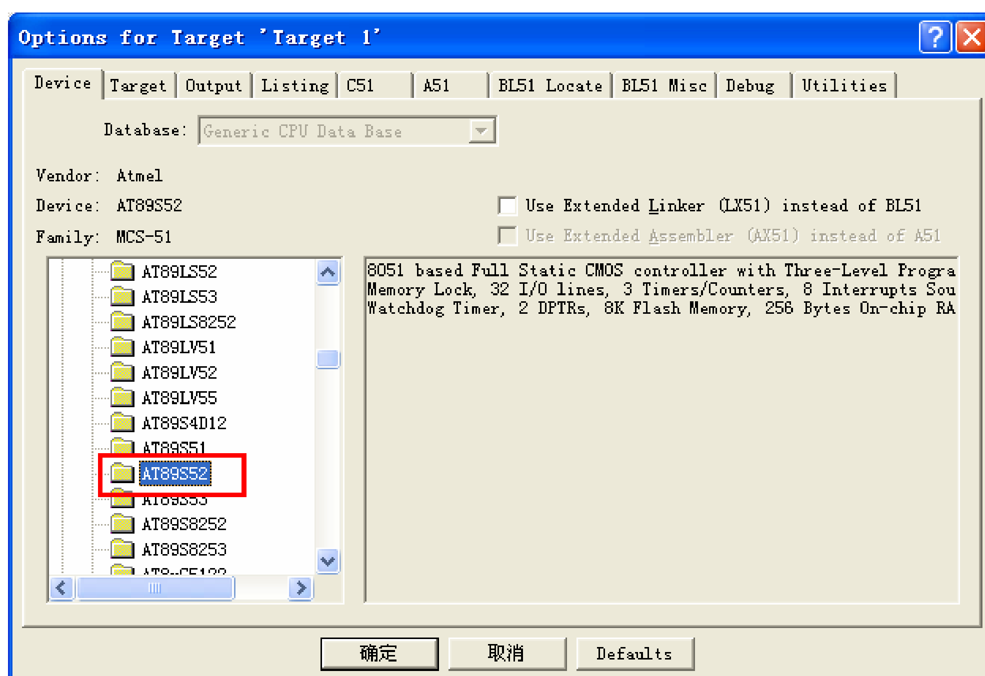


图 2-10 新建工程操作（3） 选择单片机芯片型号

4. 接下来的窗口问我们是否需要拷贝AT89S52的启动代码到工程文件中，这份启动代码在51单片机系列中都是适用的，一般情况下我们都点击是，如图2-11。



图2-11 新建工程操作（4）添加自带启动代码

5. 此时我们的工程新建成功，如下图2-12所示。但我们的工程中还没有任何文件，接下来我们需要在我们的工程中添加所需文件。

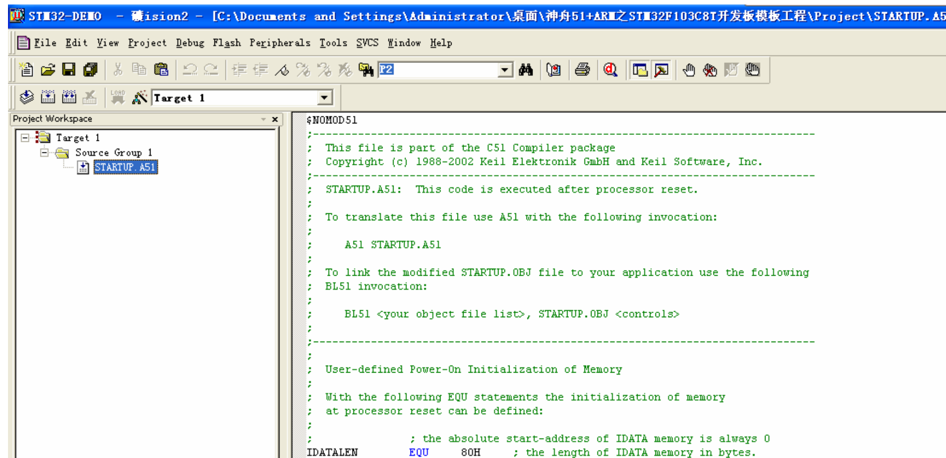


图2-12工程新建成功

7．新建一个 main.c 文件存放在路径：神舟 51+ARM 之 STM32F103C8T 开发板模板工程\Project\下，然后按照以下图 2-13 操作过程把 main.c 文件添加到工程里：

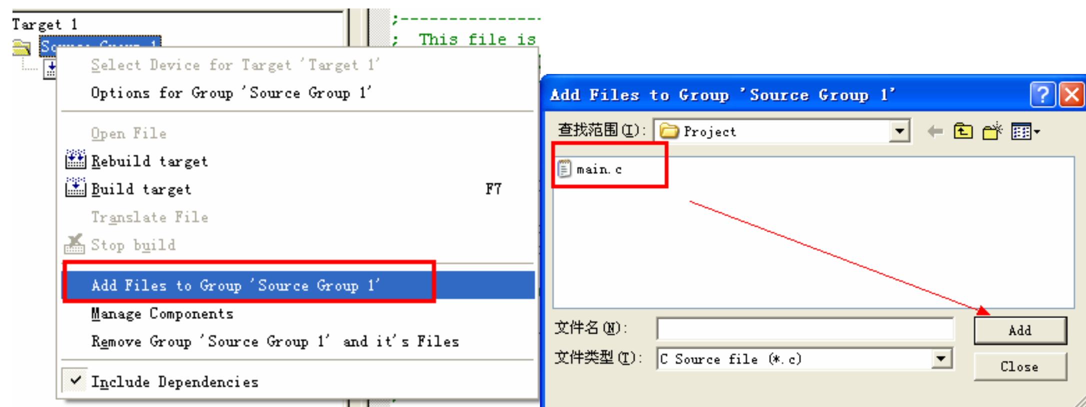


图2-13 添加工程文件

这个例程，我们将所有的代码都写到了一个 mian.c 文件，不涉及到任何库函数，也没有包含任何的头文件，如图 2-14：

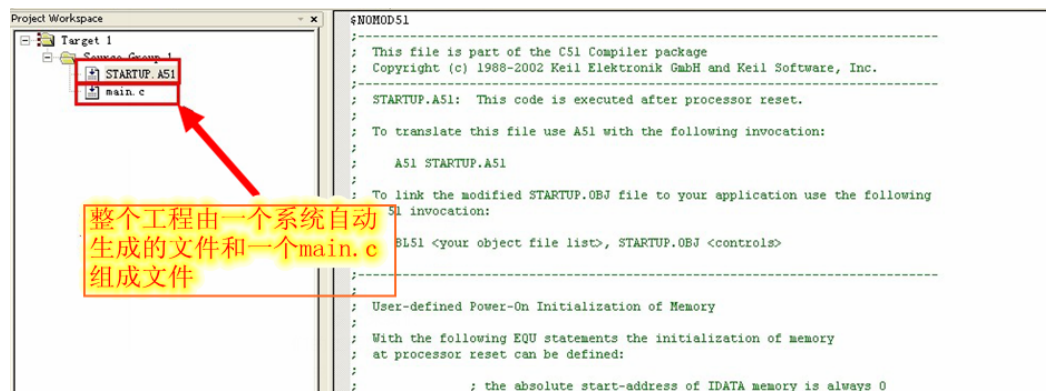


图2-14 添加Main文件

接下来我们在main.c文件上编写一个最简单的点亮一个LED灯的程序。首先是配置我们的工程编译后的存放位置，如图2-15所示：

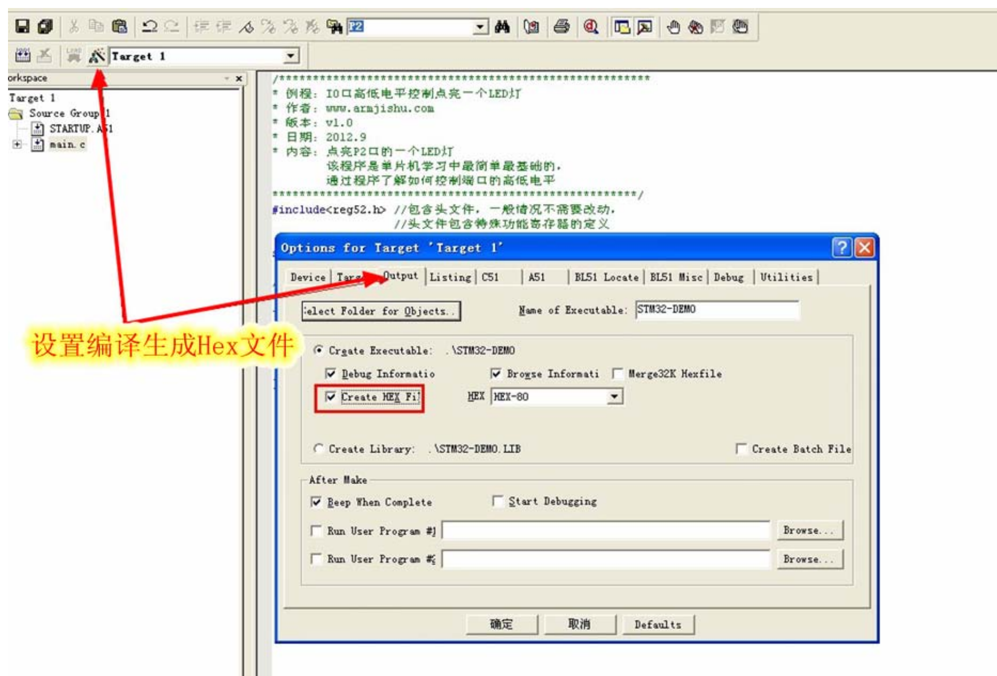


图2-15 设置编译文件存放位置

然后是编译我们的LED灯程序代码，如图2-16

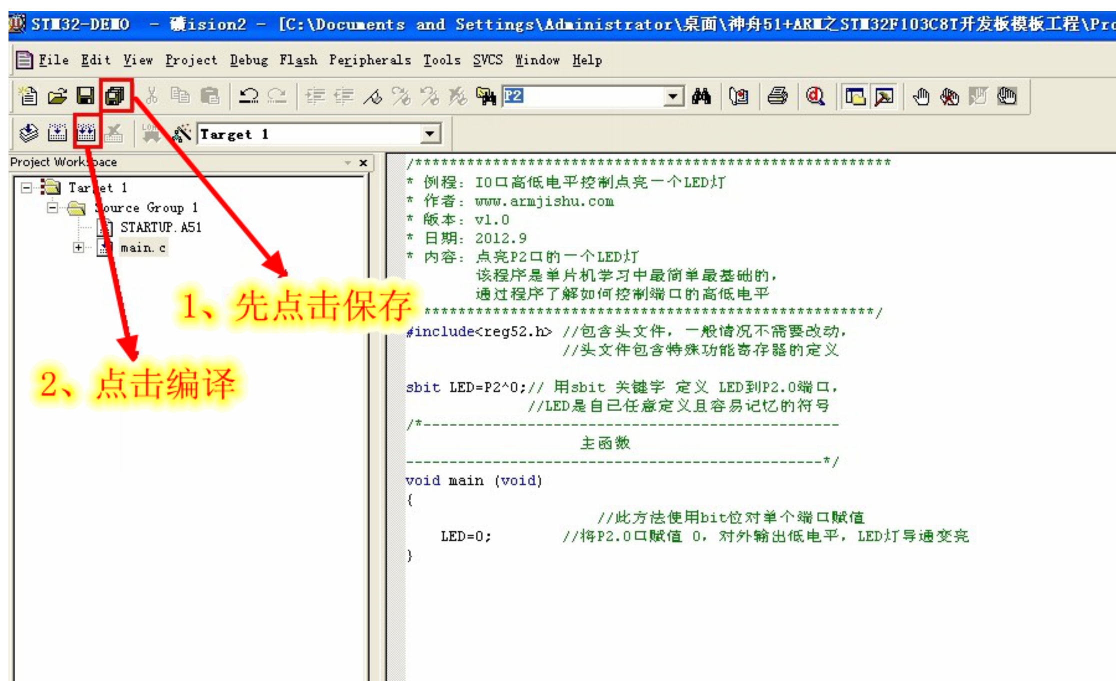


图2-16 编译程序

编译成功如图2-17



图2-17 编译成功

9．编译成功，我们可以看到编译后的 HEX 文件，直接进入 Project 文件夹，打开即可如图 2-18 和 2-19 所示：

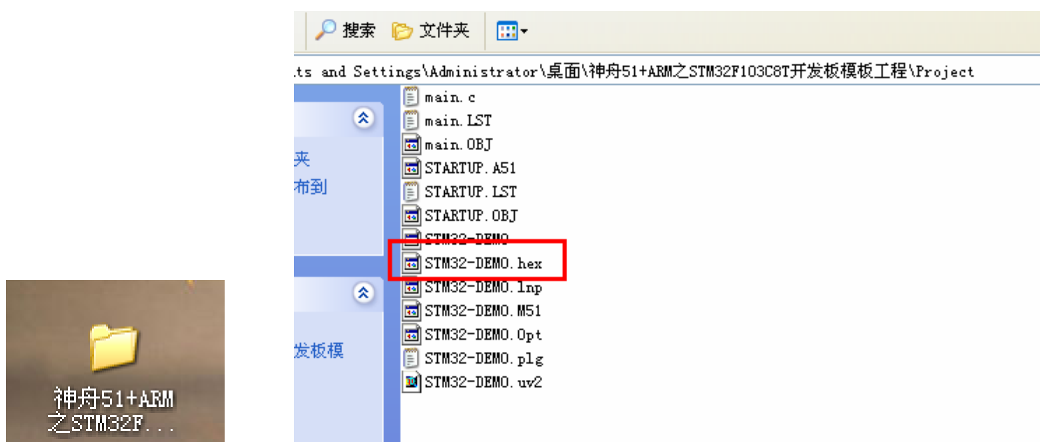


图 2-18 工程存放位置　　　　　　文件夹图 2-19 工程存放位置

10．该代码可以直接下载到神舟 51+ARM 开发板中，按一下复位按键，可以看到 LED 灯一亮一灭，具体下载方式请见其他章节

2.4.2 实现现象

可以看神舟 51+ARM 开发板的 DS1 灯一亮一灭的闪烁。

2.4.3 例程main.c源代码(可以直接运行)

```
/*******************************************************
* 例程：IO 口高低电平控制点亮一个 LED 灯
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：点亮 P2 口的一个 LED 灯
        该程序是单片机学习中最简单最基础的，
        通过程序了解如何控制端口的高低电平
*******************************************************/
#include<reg52.h> //包含头文件，一般情况不需要改动，
                  //头文件包含特殊功能寄存器的定义
```

```
sbit LED=P2^0;// 用 sbit 关键字 定义 LED 到 P2.0 端口，
                      //LED 是自己任意定义且容易记忆的符号
/*-------------------------------- 主函数 -------------------------*/
void main (void)
{
                                        //此方法使用 bit 位对单个端口赋值
      LED=0;              //将 P2.0 口赋值 0，对外输出低电平，LED 灯导通变亮
}
```

2.4.4 例程硬件原理图说明

从图 2-20 LED 的原理图中可以看到下面的 LED 灯正极是接的 VCC 端，负极接的 JP19 的座子，这个座子会与单片机的管脚相连，可以看到当 LED 灯正极是高电平的时候，只要负极为低电平，此时 LED 灯就会被点亮；那么这里只需要单片机的管脚输出为低电平，这个 LED 灯就会点亮了。

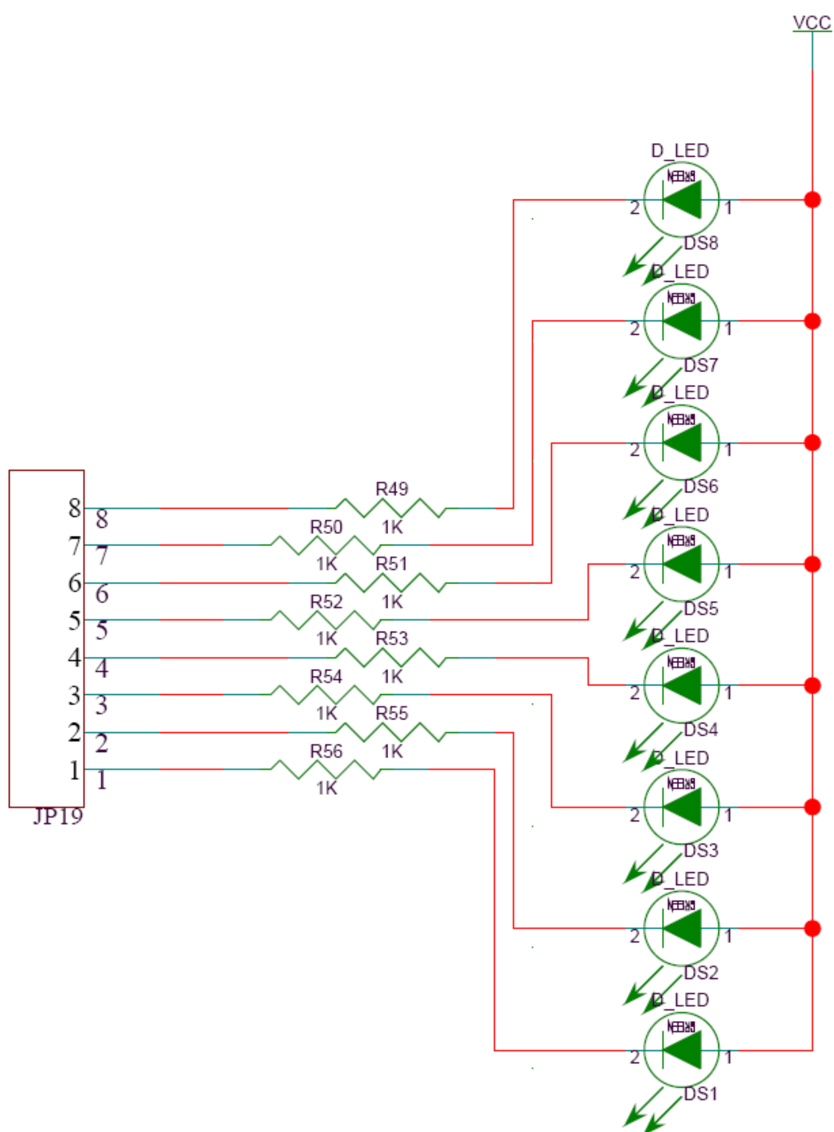


图 2-20 LED 硬件原理图

为什么这么接呢？为什么不让单片机管脚接 LED 的正极，而 LED 灯的负极去接 GND 地呢？这样才是最常规的接法不对吗？答案是当然是，但是在这里这样的接法有助于芯片的长久使用，芯片的总体驱动能力是有限的，它可以驱动一个 LED 灯，但驱动不了 100 个，1000 个。

在这里需要重复上面已经说过的内容，首先我们要知道 LED 的发光工作条件，不同的 LED 其额定电压和额定电流不同，一般而言，红或绿颜色的 LED 的工作电压为 1.7V~2.4V，蓝或白颜色的 LED 工作电压为 2.7~4.2V，直径为 3mm LED 的工作电流 2mA~10mA，在这里采用绿色的 LED；当单片机的 I/O 口作为输出口时，向外输出电流的能力是 25mA 左右，勉强是可以点亮一个发光二极管，但是如果我们用 STM32 去点亮很多个 LED 灯的时候，就有可能造成芯片本身输出电流不足(因为芯片能输出的总电流大小是恒定的)。

其次，单片机的这种接法是一种灌电流（要 VCC 往内输入电流）的方式，这种方式使得 STM32 的芯片管脚让一个 LED 灯亮非常轻松，利用灌电流的方式驱动发光二极管是比较常见的一种用法，无论接多少 LED，芯片管脚的负荷都非常轻。当然，现今的一些增强型单片机，是采用拉电流输出的，只要单片机的输出电流能力足够强即可，不过接多了也是不可取的，单片机的总体驱动电流是有限的；上图中的电阻用的是 1K 阻值主要为了限制电流，让发光二极管的工作电流限定在 2mA~10mA。

2.4.5 例程软件架构和代码分析(只有一个main.c文件)

代码如下：

```
/*****************************************************
* 例程：IO 口高低电平控制点亮一个 LED 灯
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：点亮 P2 口的一个 LED 灯
        该程序是单片机学习中最简单最基础的，
        通过程序了解如何控制端口的高低电平
****************************************************/
#include<reg52.h> //包含头文件，一般情况不需要改动，
                  //头文件包含特殊功能寄存器的定义

sbit LED=P2^0;// 用 sbit 关键字 定义 LED 到 P2.0 端口，
              //LED 是自己任意定义且容易记忆的符号
/*------------------------------------------------
                    主函数
------------------------------------------------*/
void main (void)
{
                  //此方法使用 bit 位对单个端口赋值
    LED=0;        //将 P2.0 口赋值 0，对外输出低电平，LED 灯导通变亮
}
```

分析 1：#include <reg52.h> 怎么解读，是什么意思呢？

#include <reg52.h>，# 说明这是个预处理命令（在编译之前进行的处理）， include 是文件包含命令，提示在编译的时候预先包含 reg52.h 这个文件；在代码中引用头文件，其

实际意义就是将这个头文件中的全部内容放到引用头文件的位置处，免去我们每次编写同类程序都要将头文件中的语句重复编写。

在代码中加入头文件有两种方法，分别为#include<reg52.h>和#include"reg52.h"，包含头文件时都不需要在后面加分号。两种写法区别如下：

当使用<>包含头文件时，编译器先进入到软件安装文件夹处开始搜索这个头文件，也就是Kei\C51\INC这个文件夹下，如果这个文件夹下没有用引用的头文件，编译器将会报错。

当使用双撇号""包含头文件时，编译器先进入到当前工程所在文件夹处开始搜索该头文件，如果当前工程所在文件夹下没有该头文件，编译器将继续回到软件安装文件夹处搜索这个头文件，若找不到该头文件，编译器将报错。reg52.h在软件安装文件夹存在，所以我们一般写成#include<52.h>。

打开该头文件查看内容，将鼠标移动到 reg52.h 上，单击右键，选择〖Open document<reg52.h>〗，即可打开该头文件，如图2.2.5所示。以后若需打开工程中的其他头文件，也可采用这种方法。或者手动定位到头文件所在的文件夹也可。

分析2：解析reg52.h文件

打开reg52.h文件，基本内容如下：

```
/*--------------------------------------------------------------------------
-
REG52.H
Header file for generic 80C52 and 80C32 microcontroller.
Copyright (c) 1988-2002 Keil Elektronik GmbH and Keil Software, Inc.
All rights reserved.
---------------------------------------------------------------------------*
/
#ifndef __REG52_H__
#define __REG52_H__
/*  BYTE Registers  */
sfr P0    = 0x80;
sfr P1    = 0x90;
sfr P2    = 0xA0;
sfr P3    = 0xB0;
sfr PSW   = 0xD0;
sfr ACC   = 0xE0;
sfr B     = 0xF0;
sfr SP    = 0x81;
sfr DPL   = 0x82;
sfr DPH   = 0x83;
sfr PCON  = 0x87;
sfr TCON  = 0x88;
sfr TMOD  = 0x89;
sfr TL0   = 0x8A;
sfr TL1   = 0x8B;
sfr TH0   = 0x8C;
sfr TH1   = 0x8D;
```

```
sfr IE    = 0xA8;
sfr IP    = 0xB8;
sfr SCON  = 0x98;
sfr SBUF  = 0x99;
/*  8052 Extensions  */
sfr T2CON  = 0xC8;
sfr RCAP2L = 0xCA;
sfr RCAP2H = 0xCB;
sfr TL2    = 0xCC;
sfr TH2    = 0xCD;
/*  BIT Registers  */
/*  PSW  */
sbit CY    = PSW^7;
sbit AC    = PSW^6;
sbit F0    = PSW^5;
sbit RS1   = PSW^4;
sbit RS0   = PSW^3;
sbit OV    = PSW^2;
sbit P     = PSW^0; //8052 only
/*  TCON  */
sbit TF1   = TCON^7;
sbit TR1   = TCON^6;
sbit TF0   = TCON^5;
sbit TR0   = TCON^4;
sbit IE1   = TCON^3;
sbit IT1   = TCON^2;
sbit IE0   = TCON^1;
sbit IT0   = TCON^0;
/*  IE  */
sbit EA    = IE^7;
sbit ET2   = IE^5; //8052 only
sbit ES    = IE^4;
sbit ET1   = IE^3;
sbit EX1   = IE^2;
sbit ET0   = IE^1;
sbit EX0   = IE^0;
/*  IP  */
sbit PT2   = IP^5;
sbit PS    = IP^4;
sbit PT1   = IP^3;
sbit PX1   = IP^2;
sbit PT0   = IP^1;
sbit PX0   = IP^0;
/*  P3  */
```

```
sbit RD    = P3^7;
sbit WR    = P3^6;
sbit T1    = P3^5;
sbit T0    = P3^4;
sbit INT1  = P3^3;
sbit INT0  = P3^2;
sbit TXD   = P3^1;
sbit RXD   = P3^0;
/*  SCON  */
sbit SM0   = SCON^7;
sbit SM1   = SCON^6;
sbit SM2   = SCON^5;
sbit REN   = SCON^4;
sbit TB8   = SCON^3;
sbit RB8   = SCON^2;
sbit TI    = SCON^1;
sbit RI    = SCON^0;
/*  P1  */
sbit T2EX  = P1^1; // 8052 only
sbit T2    = P1^0; // 8052 only
/*  T2CON  */
sbit TF2    = T2CON^7;
sbit EXF2   = T2CON^6;
sbit RCLK   = T2CON^5;
sbit TCLK   = T2CON^4;
sbit EXEN2  = T2CON^3;
sbit TR2    = T2CON^2;
sbit C_T2   = T2CON^1;
sbit CP_RL2 = T2CON^0;
#endif
```

从上面的代码可以看到，该头文件中定义了 52 系列单片机内部所有的功能寄存器，用到了前面讲到的 sfr 和 sbit 这两个关键字，“sfr P1=0x90;”语句的意义是，把单片机内部地址 0x90 处的这个寄存器重新起名叫 P1，以后我们在程序中可以直接操作 P1，就相当于直接对单片机内部的 0x90 地址处的寄存器进行操作。说通俗点，也就是说，通过 sfr 这个关键字，让 Keil 编译器在单片机与人之间搭建一条可以进行沟通的桥梁，我们操作的是 P1 口，而单片机本身并不知道什么是 P1 口，但是它知道它的内部地址 0x90 是什么东西。说到这里就非常清楚了，所以在编写 51 单片机程序时，我们在源代码的第一行就可以直接包含该头文件，这样就使得写起代码来比较方便一些。

在上面还可以看到，”sbit SM0 = SCON^7;”语句的意思是，将 SCON 这个寄存器的最高位第 8 位（从 0 开始到第 7，总共是 8 位）重新命名为 SM0,以后我们要单独操作 SCON 寄存器的最高位时，就可直接操作 SM0，其他都相似。

分析 3：sbit 和 sfr 关键字

sbit 这个关键字是 C51 中特有的，用于定义 SFR(特殊功能寄存器)的位变量（什么叫

位？一个字节有 8 个位变量）

例如：sbit LED=P2^0; 表示定义发光管连接的硬件端口，LED 定义在 P2（特殊功能寄存器）的第 0 位，即 P2.0，定义了这个端口以后，下面对 P2.0 的操作，我们就可以直接用 LED 代替：

```
sbit LED=P2^0;
LED=0;              //将 P2.0 口赋值 0，对外输出低电平
```

在数字电路的世界中，所有电信号都是由 0 和 1 来描述的，所以由于 sbit 定义位变量，赋值结果不是 0 就是 1。

分析 4：main()函数关键字

一个程序有且只有一个 main 函数，main()称之为主函数，是所有程序运行的入口。C 程序最大的特点就是所有的程序都是用函数来装配的;函数分为带参数的函数或不带参数的函数两种，参数均在调用的该函数的时候进行参数值的传递。

我们例程里的 main(void)加了个“void”表示 main 函数里没有任何参数，不加 void 也可以，void 表示空型。

本程序短小，但包含了一个 c 程序最基础的部分，以后的程序会在这个基础上不断添加内容以增加功能，整体结构不会改变。

分析 5： // 和 /* */ 这 2 种符号表示注释

注释不是程序，不影响程序结果，注释是给我们程序员看的，我们可以通过注释了解程序的意图，尤其在程序庞大时，注释尤为重要，如果没有注释，一段时间后，我们自己写的程序自己都看不懂了。所以养成一个好的习惯，写程序是及时注释。

上述 2 种注释符号的区别如下:

// 后面的语句都为注释，换行后无效;

/* */ 中间的内容皆为注释，换行有效。

上述样例中开头的描述使用了/* */注释，而程序中各个语句后面的注释使用了 // 。这个可以根据个人喜好，没有具体要求。

2.4.6 while语句

那么如何让程序停止在某处呢？我们用 while 语句就可以实现。

知识点：while()语句

格式：while（表达式）

{内部语句（内部可为空）}

特点：先判断表达式，后执行内部语句。

原则：若表达式不是 0，即为真，那么执行语句。否则跳出 while 语句，执行后面的语句。

需要注意的三点：

(l)在 C 语言中我们一般把“0”认为是“假”,“非 0”认为是“真”，也就是说，只要不是 0 就是真，所以 1，2，3 等都是真。

(2)内部语句可为空，就是说 while 后面的大括号里什么都不写也是可以的，如“while(l){}；”既然大括号里什么也没有，那么我们就可以直接将大括号也不写，再如“while(l);”中“；”一定不能少，否则 while()会把跟在它后面第一个分号前的语句认为它的内部语句。

例如：while(l)

P1=123;

P2=121；

….

上面这介例子中，while()会把“P1=123;”当做它的语句，即使这条语句并没有加大括号。既然如此，那么我们以后在写程序时，如果 while()内部只有一条语句，我们就可以省去大括号，而直接将这条语句跟在它的后面。

例如：while(l)

P1=123;

(3)表达式可以是一个常数、一个运算或一个带返回值的函数。

有了上面的介绍，我们在程序最后加上“while(l);’’这样一条语句就可以让程序停止。因为该语句表达式值为 1，内部语句为空，执行时先判断表达式值，因为为真，所以什么也不执行，然后再判断表达式，仍然为真，又不执行，因为只有当表达式值为 0 时才可跳出 while()语句，所以程序将不停地执行这条语句，也就是说单片机点亮发光管后将永远重复执行这条语句。

初学者可能会这样想，我让单片机把发光二极管点亮后，就让它停止工作，不再执行别的指令，这样不是更好吗？请大家注意，单片机是不能停止工作的，只要它有电，有晶振在起振，它就不会停止工作，每过一个机器周期，它内部的程序指针就要加 1，程序指针就指向下一条要执行的指令。想让它停止工作的办法就是把电断掉，不过这样发光二极管也就不会亮了。不过我们可以将单片机设置为怵眠状态或掉电模式，这样可以最大限度地降低它的功耗。关于这些内容我们在后面会讲到。

```
/*****************************************************
* 例程：IO 口高低电平控制点亮一个 LED 灯
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：点亮 P2 口的多个 LED 灯
        该程序是单片机学习中最简单最基础的，
        通过程序了解如何控制端口的高低电平
*****************************************************/
#include<reg52.h> //包含头文件，一般情况不需要改动，
                  //头文件包含特殊功能寄存器的定义
/*------------------------------------------------
                主函数
------------------------------------------------*/
void main (void)
{
   /* 该方法使用 bit 位对单个端口赋值 */
    P2=0xAA;      //换成二进制是 1010 1010
    while (1);     //主循环
}
```

2.4.7 for语句

知识点：for 语句

格式：for（表达式 1；表达式 2；表达式 3）

{语句（内部可为空）}

执行过程：

第 1 步，求解一次表达式 1。

第 2 步，求解表达式 2，若其值为真（非 0 即为真），则执行 for 中语句，然后执行第 3 步；否则结束 for 语句，直接跳出，不再执行第 3 步。

第 3 步，求解表达式 3。

第 4 步，跳到第 2 步重复执行。

需要注意的是，三个表达式之间必须用分号隔开。

利用 for 语句和 while 语句可以写出简单的延时语句，下面就用 for 语句来写一个简单的延时语句，并进一步讲解 for 语句的用法。

```
unsigned char i；
for(i=2;i>0;i--);
```

看上面这两句，首先定义一个无符号字符型变量 i，然后执行 for 语句，表达式 1 是给 i 赋一个初值 2,表达式 2 是判断 i 大于 0 是真还是假，表达式 3 是 i 自减 1，我们分析执行过程：

第 1 步，给 i 赋初值 2，此时 i=2。

第 2 步，因为 2>0 条件成立，所以其值为真，那么执衍一次 for 中的语句，因为 for 内部

语句为空，即什么也不执行。

第 3 步，i 自减 1，即 i=2-1=1。

第 4 步，跳到第 2 步，因为 1>0 条件成立，所以其值为真，那么执行一次 for 中的语句，因为 for 内部语句为空，即什么也不执行。

第 5 步，i 自减 1，即 i=1-1=0。

第 6 步，跳到第 2 步，因为 0>0 条件不成立，所以其值为假，那么结束 for 语句，直接跳出。

通过以上 6 步，这个 for 语句就执行完了，单片机在执行这个 for 语句的时候是需要时间的，上面 i 的初值较小，所以执行的步数就少，当我们给 i 赋的初值越大，它执行所需的时间就越长，因此我们就可以利用单片机执行这个 for 语句的时间来作为一个简单延时语句。

很多初学者容易犯的错误是，想用 for 语句写一个延时比较长的语句，那么他可能会这样写：

```
unsigned char i；
for(i=3000;i>0;i--);
```

但是结果却发现这样写并不能达到延长时间的效果，因为在这里 i 是一个字符型变量，它的最大值为 255，当你给它赋一个比最大值都大的数时，编译器自然就出错误了，因此我们尤其要注意，每次给变量赋初值时，都要首先考虑变量类型，然后根据变量类型赋一个合理的值。

那么怎样才能写出长时间的延时语句呢？我们下面讲解 for 语句的嵌套。

```
unsigned char i,j;
for(i=100;i>0;i--)
for(j=200;j>0;j--);
```

上面这个例子是 for 语句的两层嵌套，大家注意看，第一个 for 后面没有分号，那么编译器默认第二个 for 语句就是第一个 for 语句的内部语句，而第二个 for 语句内部语句为空，

程序在执行时，第一个 for 语句中的 i 每减一次，第二个 for 语句便执行 200 次，因此上面这个例子便相当于共执行了 100×200 次 for 语句。通过这种嵌套我们便可以写出比较长时间的延时语句，我们还可以进行 3 层、4 层嵌套来增加时间，或是改变变量类型，将变量初值再增大也可以增加执行时间。

这种 for 语句的延时时间到底有没有精确的算法呢？在 C 语言中这种延时语句不好算出它的精确时间，如果需要非常精确的延时时间，我们在后面会讲到利用单片机内部的定时器来延时，它的精度非常高，可以精确到微秒级。而一般的简单延时语句实际上我们并不需要太精确，不过我们也是有办法知道它大概延时多长时间的，请看下一节讲解。

2.5 KEIL仿真及延时语句的精确计算

在这里贴出一段代码来进行深入的分析：

```
#include <reg52.h>                          //52 系列单片机头文件
#define uint unsigned int                   //宏定义
sbit led1=P2^0;                             //声明单片机 P1 口的第一位
uint i,j;
void   main()                               //主函数
{
    while(1)                                //大循环
    {
        led1=0;                             /*点亮第一个发光二极管*/
        for(i=1000;i>0;i--)                 /*延时*/
            for(j=110;j>0;j--);
                led1=1;                     /*关闭第一个发光二极管*/
        for(i=1000;i>0;i--);                /*延时*/
            for(j=110;j>0;j--);
    }
}
```

我们如何用软件来模拟出这个延时语句究竟是延长多少时间呢？回到 Keil 编辑界面，打开工程设置对话框，在【Target】习标签下的【Xtal(MHz)】：把后面将原来的默认值修改为神舟 51+ARM 单片机实验板上晶振频率值 11.0592MHz，如下图 2-21 所示：

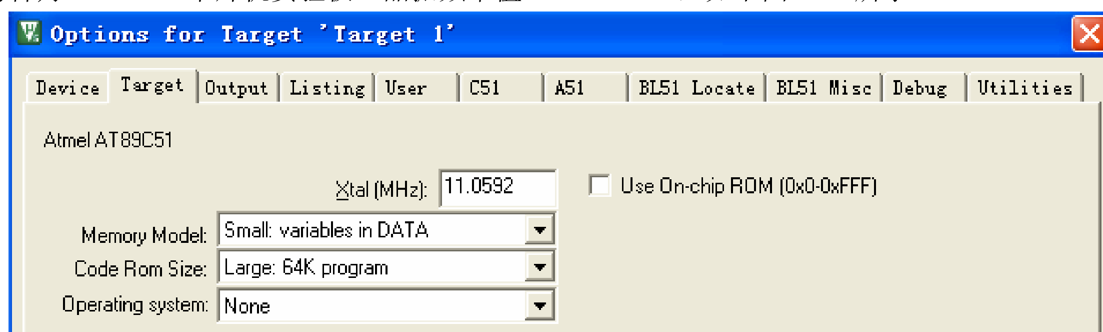


图 2-21 KEIL 晶振配置

Keil 编译器在编译程序时，计算代码执行时间与该数值有关，既然我们要模拟真实时间，那么软件模拟运行速度就要与实际硬件一一对应，神舟 51+ARM 单片机上使用的外部晶振频率是 11.0592MHz，在单片机的中下方大家可以看到实物，如下图 2-22 所示。



图 2-22 神舟 51 单片机晶振实物图

单击【确定】按钮后，再单击窗口上的调试按钮快捷图标，进入到软件模拟调试模式，如下图 2-23 所示：

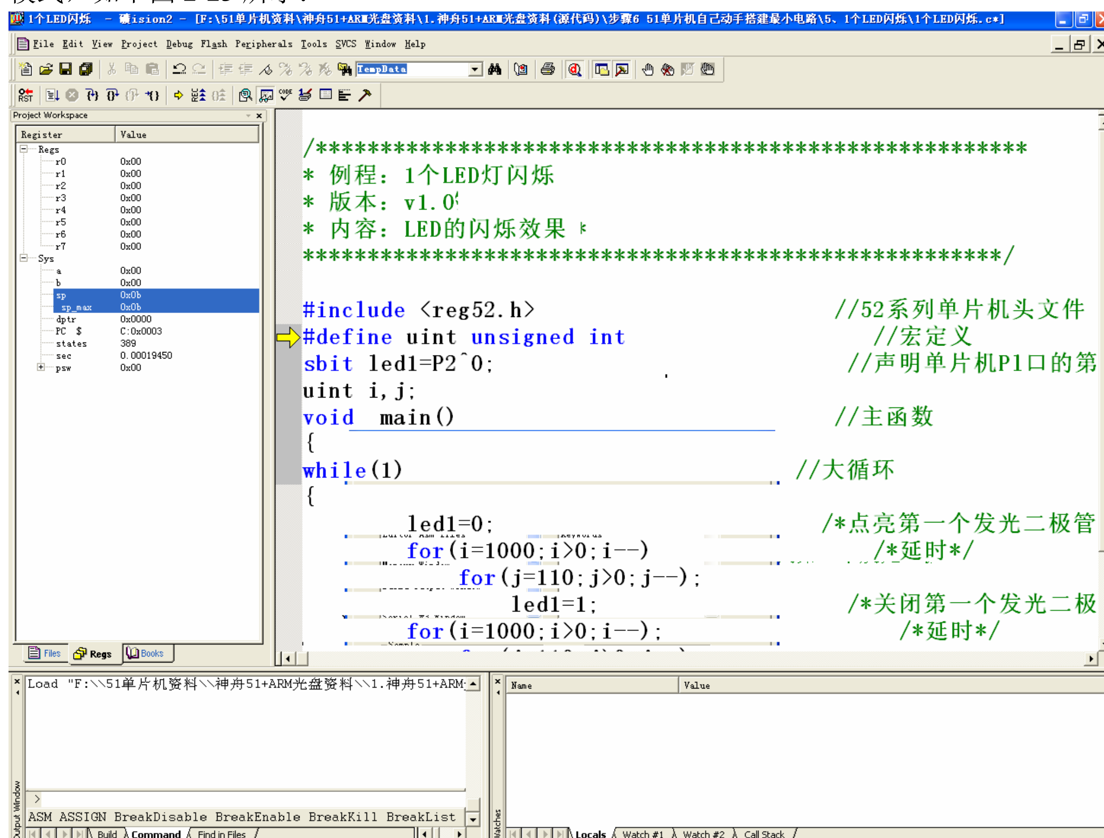


图 2-23 模拟调试模式

可以看到，在上图左侧的寄存器窗口中可以看到一些寄存器名称及它们的值，如下图 2-24 所示：

Project Workspace

Register	Value
Regs	
r0	0x00
r1	0x00
r2	0x00
r3	0x00
r4	0x00
r5	0x00
r6	0x00
r7	0x00
Sys	
a	0x00
b	0x00
sp	0x0b
sp_max	0x0b
dptr	0x0000
PC $	C:0x0003
states	389
sec	0.00014145
psw	0x00

图 2-24　各寄存器的值

在软件调试模式下，我们可以设置断点、单步、全速、进入某个函数内部运行程序，同时还可以查看变量变化过程、模拟硬件 I/O 口电平状态变化、查看代码执行时间等。在开始调试之前我们先熟悉一下调试按钮的功能。调试状态下多了图 2-25 所示的几个调试按钮。

图 2-25 调试按钮

对常用的几个按钮介绍如下：

RST——将程序复位到主函数的最开始处，准备重新运行程序。

——全速运行，运行程序时中间不停止。

——停止全速运行，全速运行程序时激活该按钮，用来停止正全速运行的程序。

——进入子函数内部。

——单步执行代码，它不会进入子函数内部，可直接跳过函数。

——跳出当前进入的函数，只有进入子函数内部该按钮才被激活。

——程序直接运行至当前光标所在行。

——显示 / 隐藏编译窗口，可以查看每句 C 语言编译后所对应的汇编代码。

——显示 / 隐藏变量观察窗口，可以查看各个变量值的变化状态。

大家不妨把这些按钮一个个都单击试试看，只有亲自操作过了记忆才会深刻。我们先

来看如何在单步执行代码时，查看硬件 I/O 口电平变化和变量值的变化。先将硬件 I/O 口模拟器打开，在图 2-26 中单击【Port 2】项，弹出下图 2-27 所示对话框。

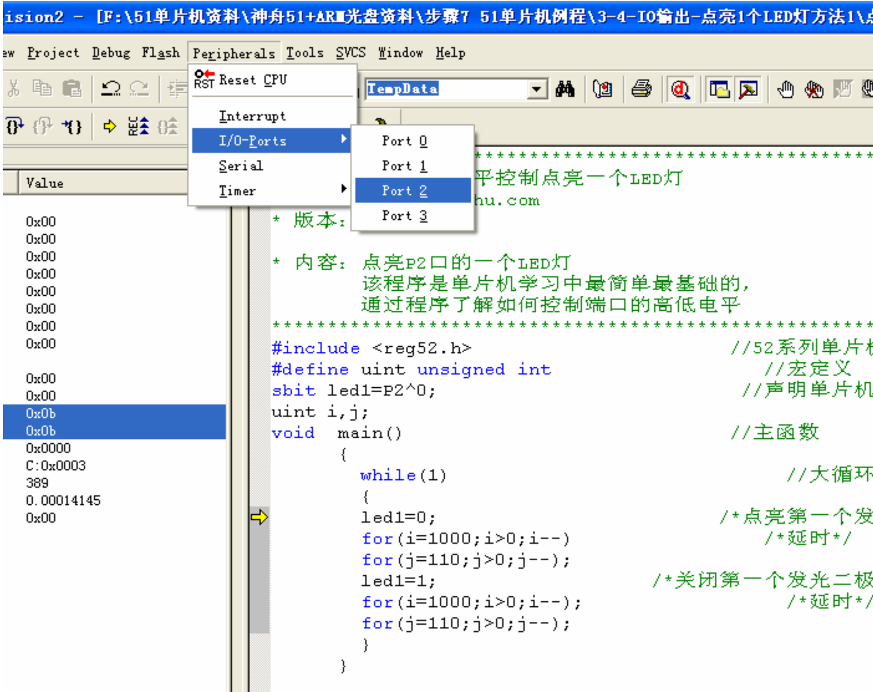


图 2-26 查看硬件 I/O 口操作

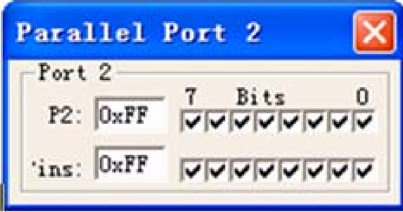


图 2-27 查看硬件 I/O 口

图 2-27 中显示的是软件模拟出的单片机 Pl 口 8 位口线的状态，单片机上电后 I/O 口全为 1，即十六进制的 0xFF。

找们再单击图 2-28 中右下角变量观察窗口的【Watch# 1】标签，可以看到上面显示出“type F2 to edit（按 F2 进行编辑）”的字样

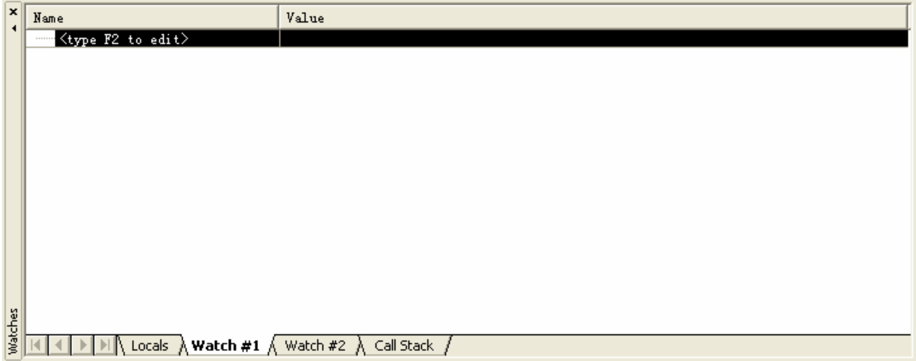


图 2-28 变量观察窗口（1）

接下来我们分别按两次 F2，输入本程序中用到的两个变量 i 和 j。在右面立即显示出变量的值 0x0000，如下图 2-29 所示，因为 i 和 j 在最开始定义的时候并没有给它们赋初值，编译器默认给它们赋的初值是 0，而当进入 for 语句后，我们才为 i 和 j 分别赋了 1000 和 110

的值。

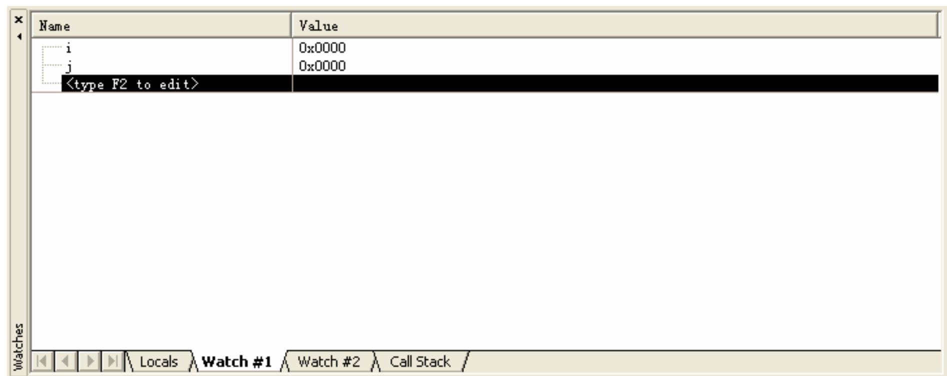


图 2-29 变量观察窗口（2）

现在我们最关心的只有一个，就是图 2-30 中左侧的寄存器一个叫“sec”的部分：

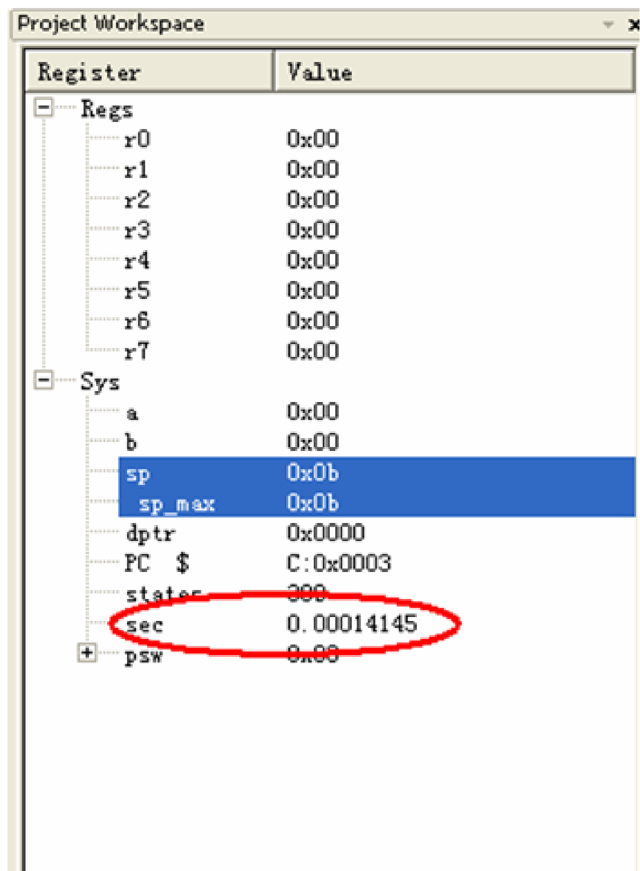


图 2-30 寄存器“sec”

它后面显示的数据就是程序代码执行所用的时间，单位是秒，这是程序启动执行到目前停止位置所花的所有时间。注意：这个时间是累计时间。

我们回到代码编辑框，看到主函数“ledl=0;”前面有一个黄色的小箭头，如图 2-31 所示：

```
/************************************************************
* 例程：IO口高低电平控制点亮一个LED灯
* 作者：www.armjishu.com
* 版本：v1.0

* 内容：点亮P2口的一个LED灯
        该程序是单片机学习中最简单最基础的，
        通过程序了解如何控制端口的高低电平
**********************************************************/
#include <reg52.h>                          //52系列单片机头文件
#define uint unsigned int                     //宏定义
sbit led1=P2^0;                            //声明单片机P1口的第一位
uint i,j;
void  main()                               //主函数
      {
        while(1)                                 //大循环
        {
        led1=0;                          /*点亮第一个发光二极管*/
        for(i=1000;i>0;i--)                  /*延时*/
        for(j=110;j>0;j--);
        led1=1;                     /*关闭第一个发光二极管*/
        for(i=1000;i>0;i--);                  /*延时*/
        for(j=110;j>0;j--);
        }
      }
```

图 2-31 程序执行位置

这里需要注意，这个小箭头指向的代码是下一步将要执行的代码，我们单击单步运行快捷图标 {}，这时看到黄色小箭头向下移动了一行，在 Pl 口软件模拟窗口中，Pl 的最低位对应的对号没有了，这说明“led2=0;”这条语句执行结束了，在实际硬件中也就点亮了 P2 口最低位所对应的发光二极管。同时 sec 后面变成为 323.18 us，因为我们可以计算出执行这条指令实际花去了 323.18-322.09=1.091us 的时间，这个时间恰好就是 51 单片机在 11.0592 晶振频率下，一个机器周期所花费的时间。

接着再单击单步运行按钮，这时右下角变量查看窗口中的 i 被赋值 0x03E8，在这个值上单击鼠标右键选择【Number Base→Decimal】项，将数值显示方式改成十进制显示，我们看到 i 的值即为 1000，实际上就是刚才上一步运行第一个 for 语句时给 i 赋的值。继续单步运行可以看到 i 的值从 1000 开始往下递减，同时左侧的 sec 在一次次增加，但 j 酌值始终为 O，因为每执行一次外层 for 语句，内层 for 语句将执行 110 次，即 j 已经由 110 递减到 O 了，所以我们看上去 j 的值始终都是 0。那如果我们要看这个 for 嵌套语句到底执行了多长时间的话，是不是就要单击 1000 次呢？其实不用这么麻烦，设置断点可以方便地解决这个问题。

设置断点有很多好处，在软件模拟调试状态下，当程序全速运行时，每遇到断点，程序会自动停止在断点处，即下一步将要执行断点处所在的这条指令。这样，我们只需在延时语句的两端各设置一个断点，然后通过全速运行，便可方便地计算出所求延时代码的执行时间。设置方式如下：单击复位钮，然后在第一个 for 所在行前面空白处双击鼠标，前面出现一个红色方框，表示本行设置了一个断点，然后在下面“ledl=l;”所在行以同样方式插入另一个断点，这两个断点之间的代码就是这个两级 for 嵌套语句，如下图 2-32 所示。

```
/*********************************************************
* 例程：IO口高低电平控制点亮一个LED灯
* 作者：www.armjishu.com
* 版本：v1.0

* 内容：点亮P2口的一个LED灯
        该程序是单片机学习中最简单最基础的，
        通过程序了解如何控制端口的高低电平
*********************************************************/
#include <reg52.h>                          //52系列单片机头文件
#define uint unsigned int                      //宏定义
sbit led1=P2^0;                              //声明单片机P1口的第一位
uint i,j;
void  main()                                //主函数
      {
        while(1)                                  //大循环
        {
        led1=0;                            /*点亮第一个发光二极管*/
        for(i=1000;i>0;i--)                    /*延时*/
        for(j=110;j>0;j--);
        led1=1;                      /*关闭第一个发光二极管*/
        for(i=1000;i>0;i--);                      /*延时*/
        for(j=110;j>0;j--);
        }
      }
```

2-32 断点设置

单击全速运行按钮，程序会自动停止在第一个 for 语句所在行，查看时间显示为 323.181us，再单击一次全速运行按钮，程序停止在第二个 for 语句下面一行处，查看时间显示为 868.31272ms，我们忽略微秒，此时间约为接近 1s，由于无须精确时间，所以这个精度已经足够，我们的 for 语句延时时间便计算出来了。

大家可以改变 for 语句中两个变量的初值来重新测试时间， for 语句中两个变量类型都为 unsigned int 型时（注意，若变量为其他类型则时间不遵循以下规律，因为变量类型不同，单片机运行时所需时间就不同），内层 for 语句中变量恒定值为 110 时，外层 for 中变量为多少，这个 for 嵌套语句就延时约多少毫秒，大家可自行测试验证，也可自己测试出更精确的延时语句。

2.6 不带参数函数的写法及调用

可以看到在打开和关闭发光二极管的两条语句之后，是两个完全相同的 for 嵌套语句：

```
for(i=l000;i>0;i--);
for(j=110;j>0;j--);
```

在 C 语言代码中，如果有一些语句不止一次用到，而且语句内容都相同，我们就可以把这样的一些语句写成一个不带参数的子函数，当在主函数中需要用到这些语句时，直接调用这个子函数就可以了。我们以上面这个 for 嵌套语句为例，其写法如下：

```
void delayls()
{
    for(i=l000;i>0;i--)
    for(j=110;j>0;j--);
}
```

其中，void 表示这个函数执行完后不返回任何数据，即它是一个无返回值的函数。delayls

是函数名，这个名字我们可以随便起，但是注意不要和 C 语言中的关键字相同。大家写成 delay_ls，delaylmiao 等都是可以的，一般我们写成方便记忆或读懂的名字，也就是一看到函数名就知道此函数实现的内容是什么。我在这里写成 delayls 是因为这个函数是一个延时 1s 的函数。紧跟函数名后面的是一个括号，这个括号里没有任何数据或符号（即 C 语言当中的“参数”），因此这个函数是一个无参数的函数。接下来两个大括号中包含着其他要实现的语句。以上讲解的是一个无返回值、不带参数的函数的写法。

需要注意的是，子函数可以写在主函数的前面或是后面，但是不可以写在主函数里面。当写在后面时，必须要在主函数之前声明子函数，声明方法如下：将返回值特性、函数名及后面的小括号完全复制，若是无参函数，则小括号内为空；若是有参函数，则需要在小括号里依次写上参数类型，只写参数类型，无须写参数，参数类型之间用逗号隔开。最后在小括号的后面必须加上分号“；”。当子函数写在主函数前面时，不需要声明，因为写函数体的同时就已经相当于声明了函数本身。通俗地讲，声明子函数的目的是为了编译器在编译主程序的时候，当它遇到一个子函数时知道有这样一个子函数存在，并且知道它的类型和带参情况等信息，以方便为这个子函数分配必要的存储空间。

```
#include <reg52.h>                         //52 系列单片机头文件
#define uint unsigned int                  //宏定义
sbit led1=P2^0;                            //声明单片机 Pl 口的第一位
int i,j;
void delay_shenzhou();x
void   main()                              //主函数
{
     while(1)                              //大循环
     {
          led1=0;                          /*点亮第一个发光二极管*/
          delay_shenzhou();
          led1=1;                          /*关闭第一个发光二极管*/
          delay_shenzhou();
     }
}
Void delay_shenzhou()
{
          Int I,j;
          for(i=1000;i>0;i--);
               for(j=110;j>0;j--);

}
```

我们注意到“uint i，j；”语句，i 和 j 两个变量的定义放到了子函数里，而没有写在主函数的最外面。在主函数外面定义的变量叫做全局变量；像这种定义在某个子函数内部的变量被叫做局部变量，这里 i 和 j 就是局部变量。注意：局部变量只在当前函数中有效，程序一旦执行完当前子函数，在它内部定义的所有变量都将自动销毁，当下次再调用该函数时，编译器重新为其分配内存空间。我们要知道，在一个程序中，每个全局变量都占据着单片机内固定的 RAM，局部变量是用时随时分配，不用时立即销毁。一个单片机的 RAM 是有限的，如 AT89C52 只有 256B 的 RAM，如果要定义 unsigned int 型变量的话，我们最多只能定义 128 个；STC 单片机内部比较多，有 512B 的，也有 1280B 的。很多时候，当写一个比较

大的程序时，经常会遇到内存不够用的情况，因此我们从一开始写程序时就要坚持能节省RAM 空间就要节省，能用局部变量就不用全局变量的原则。

将程序下载到实验板，可看见小灯先亮 500ms，再灭 500ms，一直闪烁。

2.7 带参数函数的写法及调用

Delay_shenzhou()子函数，i=500 时延时 500ms，那么如果我们要延时 300ms，就需要在子函数里把 i 再赋值为 300，要延时 l00ms 就得改 i 为 100，这样岂不是很麻烦？有了带参数的子函数就好办多了，写法如下：

```
void delay_shenzhou(unsigned int xms)
{
      uint i,j；
      for(i=xms;i>0;i--)       //i=xms 即延时约 xms 毫秒
      for(,j=ll0;j>0;j--);
}
```

上面代码中 delay_shenzhou 后面的括号中多了一句“unsigned int xms”，这就是这个函数所带的一个参数，xms 是一个 unsigned int 塑变量，又叫这个函数的形参，在调用此函数时我们用一个具体真实的数据代替此形参，这个真实数据被称为实参，形参被实参代替之后，在子函数内部所有和形参名相同的变量将都被实参代替。声明方法在前面已经讲过，这里再强调一下，声明时必须将参数类型带上，如果有多个参数，多个参数类型都要写上，类型后面可以不跟变量名，也可以写上变量名。有了这种带参函数，我们要调用一个延时 300ms 的函数就可以写成“delay_shenzhou (300);”，要延时 200ms 可以写成“delay_shenzhou (200);”，这样就方便多了。如下：

写一个完整的程序，还是让一个小灯闪动，不过这次我们让它以亮 100ms、
灭 900ms 的方式闪动，完整程序代码如下：

```
#include <reg52.h>                          //52 系列单片机头文件
#define uint unsigned int                   //宏定义
sbit led1=P2^0;                              //声明单片机 Pl 口的第一位
int i,j;
void delay_shenzhou(int);
void   main()                                //主函数
{
      while(1)                                  //大循环
      {
            led1=0;                          /*点亮第一个发光二极管*/
            delay_shenzhou(100);
            led1=1;                         /*关闭第一个发光二极管*/
            delay_shenzhou(900);
      }
}
Void delay_shenzhou(int xms)
{
            Int I,j;
```

```
        for(i=xms;i>0;i--);
            for(j=110;j>0;j--);

}
```

将程序下载到实验板，可看见小灯先亮 100sm，再灭 900ms，一直闪烁。

2.8 利用C51 库函数实现流水灯

实现流水灯的办法有多种，可以用逻辑运算来实现，也可以用 C51 库自带的函数来实现，本节中我们就调用现成的库函数来实现流水灯，大家打开 Keil 软件安装文件夹，定位到 Keil\C51\HLP 文件夹，打开此文件夹下的 C51 lib 文件，这是 C51 自带库函数帮助文件。在索引栏我们找到_crol_函数，双击打开它的介绍，内容如下图 2-33：

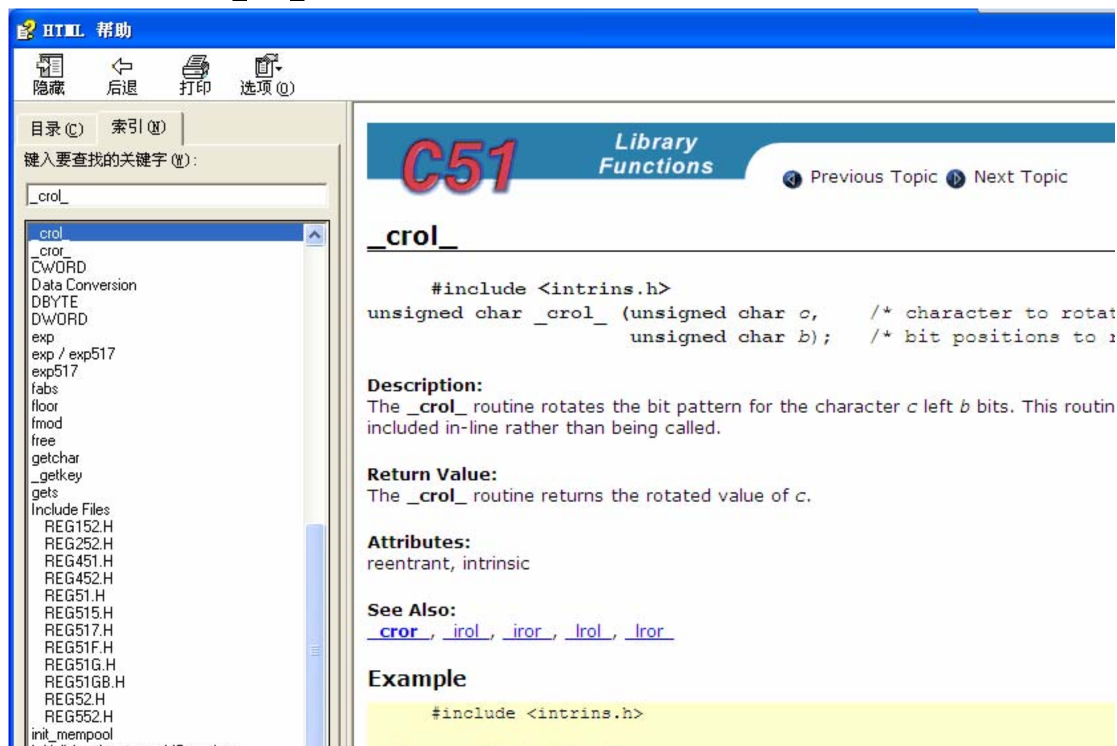


图 2-33 _crol_函数

这个函数包含在 intrins.h 头文件中，也就是说，如果在程序中要用到这个函数，那么必须在程序的开头处包含 intrins.h 这个头文件。再来看函数特性“unsigned char _crol_ (unsigned char c，unsigned char b);” 这个函数不像前几节我们讲过的函数，它前面没有 void，取而代之的是 unsigned char;小括号里有两个形参，unsigned char c，unsigned char b，这种函数叫做有返回值、带参数的函数。有返回值的意思是说，程序执行完这个函数后，通过函数内部的某些运算而得出一个新值，该函数最终将这个新值返回给调用它的语句。_crol_是函数名，不再多讲。我们再来看看函数实现了什么功能。

上面英文的大意是，Description（描述）：_crol_这个函数的意思是将字符 c 循环左移 b 位，这是 C51 库自带的内部函数，在使用这个函数之前，需要在文件中包含它所在的头文件。再看后面的 Return Value（返回值）：_crol_这个函数返回的是将 c 循环左移之后的值。关于移位操作，我们看下一个知识点。

知识点：移位操作

(1)左移。C51 中操作符为“<<”，每执行一次左移指令，被操作的数将最高位移入单片机 PSW 寄存器的 CY 位，CY 位中原来的数丢弃，最低位补 0，其他位依次向左移动一位，如下图 2-34 所示：

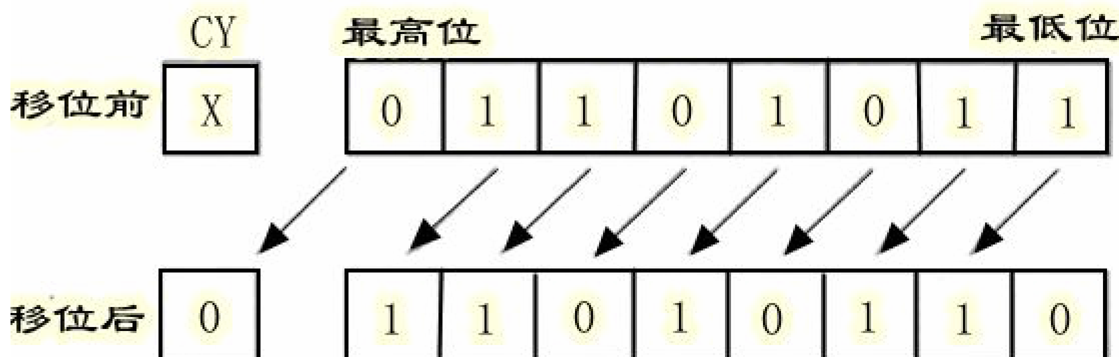


图 2-34 左移示意图

(2)右移。C51 中操作符为“>>”，每执行一次右移指令，被操作的数将最低位移入单片机 PSW 寄存器的 CY 位，CY 位中原来的数丢弃，最高位补 0，其他位依次向右移动一位，如下图 2-35 所示：

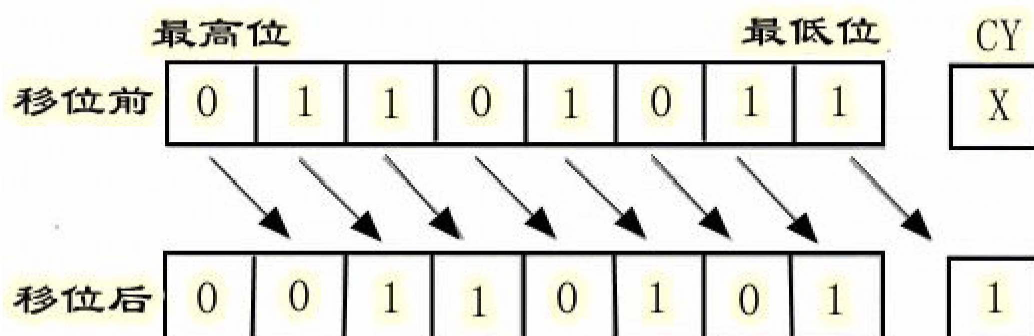


图 2-35 右移示意图

(3)循环左移。最高位移入最低位，其他位依次向左移一位。C 语言中没有专门的指令，通过移位指令与简单逻辑运算可以实现循环左移，或直接利用 C51 库中自带的函数_crol_实现，如下图 2-36 所示：

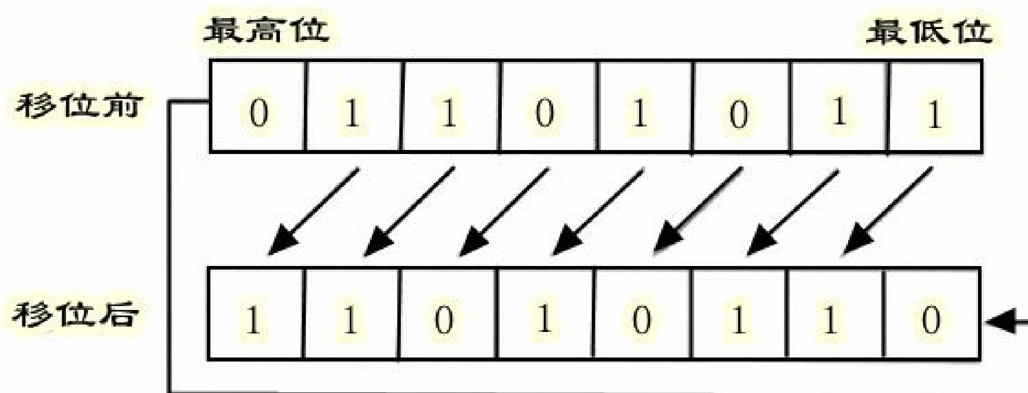


图 2-36 循环左移示意图

(4)循环右移。最低位移入最高位，其他位依次向右移一位。C 语言中没有专门的指令，

通过移位指令与简单逻辑运算可以实现循环右移，或直接利用 C51 库中自带的函数_cror_实现，如下图 2-37 所示：

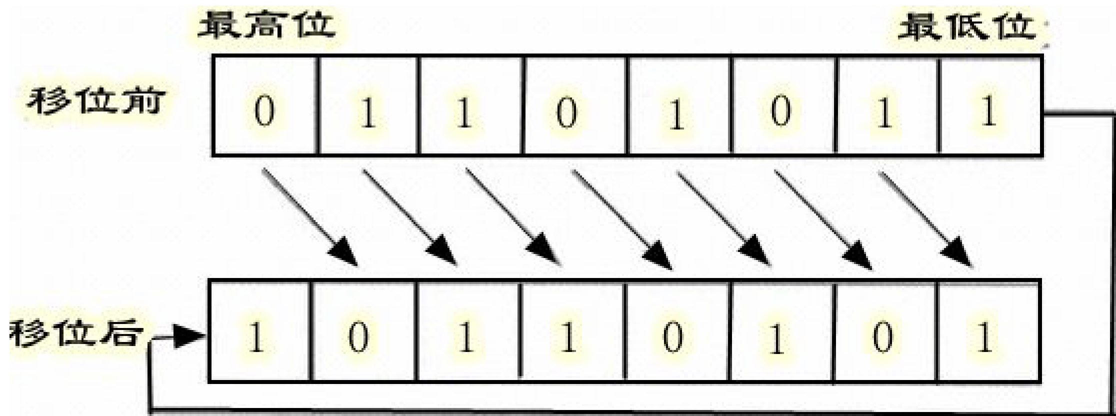


图 2-37 循环右移示意图

为了加深印象，大家可以利用软件模拟的方法在 Keil 中亲自操作一下，通过单步运行查看变量，左移、右移时可看到 PSW 寄存器中 CY 的变化与被移位数的变化状态：

```
#include <reg52.h>                          //52系列单片机头文件
#define uchar unsigned char                 //宏定义
uchar a;
void main()                                 //主函数
{
    a=0xaa;
    while(l)                                //大循环
    {
        a=a>>1;
    }
}
```

知识点：PSW 寄存器。

PSW(Program Status Word)全称为程序状态字标志寄存器，是一个 8 位寄存器，位于单片机片内的特殊功能寄存器区，字节地址 D0H，用来存放运算结果的一些特征，如有无进位、借位等，使用汇编编程时 PSW 寄存器很有用，但在利用 C 语言编程时，编译器会自动控制该寄存器，很少人为操作它，大家只需做简单了解即可。其每位的具体含义如图 2-38 所示。

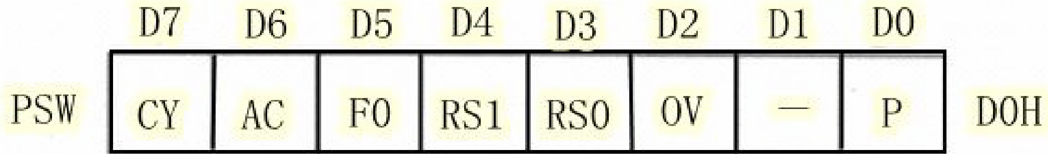


图 2-38 PSW(Program Status Word)寄存器

① **CY--**进位标志位，它表示运算是否有进位（或借位）。如果操作结果在最高位有进位（加法）或者借位（减法），则该位为 1，否则为 0。

② **AC--**辅助进位标志，又称半进位标志，它指两个 8 位数运算低四位是否有半进位，

即低四位相加（或相减）是否进位（或借位），如有 AC 为 1，否则为 0。

③ **F0--**由用户使用的一个状态标志位，可用软件来使它置 1 或清 0，也可由软件来测试它，以控制程序的流向。

④ **RS1，RS0--**4 组工作寄存器区选择控制位，在汇编语言中这两位用来选择 4 组工作寄存器区中的哪一组为当前工作寄存区。

⑤ **OV--**溢出标志位，反映带符号数的运算结果是否有溢出。有溢出时，此位为 1，否则为 0。

⑥ **P--**奇偶标志位，反映累加器 ACC 内容的奇偶性，如果 ACC 中的运算结果有偶数个 l（如 11001100B，其中有 4 个 1），则 P 为 0，否则 P 为 1。

至此，我们已经从最简单的建立工程开始，通过一步步的操作，为大家详细介绍了设计一个完整的流水灯程序的过程，从中我们学到了 Keil 软件的使用、调试模式下的软件仿真、while 语句、for 语句、各种函数的写法及用法，本章的知识非常重要，属于基础入门级讲解，大家若有不明白之处要多看几遍，多查找相关资料，最重要的是多实践，多操作，以实践与理论相结合的方式来学习，真正的将单片机及电子方面的知识全部吸收消化。

2.9 硬件基础

通过前面的学习，大家对这个单片机有了一个初步的认识。而且对这个 KEIL 软件的基本操作都有了基本的动手能力，刚开始可能会遇到一些小挫折，没关系，慢慢来，等你编程动了，就会熟能生巧，软件操作没什么。上节课主要讲了点亮一个 LED 小灯。对于单片机的 C 语言也有了一个初步的认识，现在我们讲一个单片机的硬件基础，因为我们在学习单片机 C 语言，不仅仅只是学习编程就可以的了。我们既要设计点亮，又要编写程序。设计电路的时候，我们编写程序要按照硬件电路图来。因此硬件的学习是必不可少的，这一节我们就来学下硬件电路的知识。

2.9.1 电磁干扰

1．静电放电（ESD）干扰。

在冬天的时候，空气比较干燥的城市，有时候晚上脱毛衣可以看到啪啪的响，这个就是静电，还有各种电器比如电脑，铁柜等放电，这些都叫“静电放电（ESD）干扰”。

2．快速瞬间脉冲群（EFT）的干扰

比如使用电钻的时候听收音机有杂音，看电视图像有波纹出现，这就是“快速瞬间脉冲群（EFT）的干扰”。

3.浪涌（Surge）的干扰。

浪涌电流是指的超出正常工作电流的瞬间过电流，电涌是指“常规”电压的增加，通常由剧烈变动或电力需求的增加而引起。打开大功率电器、吸尘器、空调、洗衣机都可以引发电涌和峰值电压。

2.9.2 去耦电容

通过实践来学习是最快的，所以讲去耦电容就拿神舟 51 开发板的原理图来做实际讲解，在下图 2-39 的原理图上：

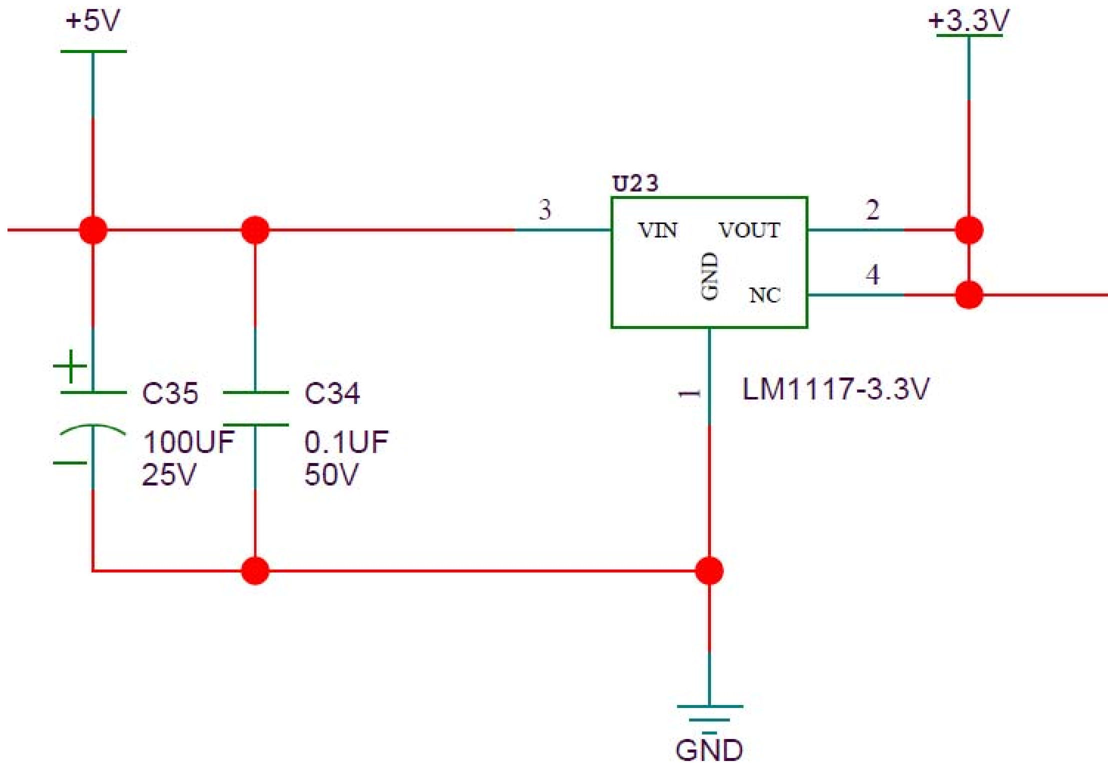


图 2-39 神舟 51 单片机开发板电路部分原理图

可以看到，这里有 1 个 100uf 的电容，一个 0.1uF 的电容，这 2 个电容是被用来静电放电的；举个简单的例子，静电放电表面看来没什么，打到身上也不怎么疼，但是它的电压是非常高的。像我们手上能感觉的静电都到 2、3 千伏左右了，如果能看得见的，都到 4、5 千伏了。它的电压都是非常高的，因为它的电量非常小，它电不到人，打到人也只是有感觉而已。但对于半导体器件就不一样了，抗高压的能力非常弱，比较严重的是损坏元器件。我们现在的特别是芯片防静电的能力都非常强了，所以说很少能打坏器件。但是在一些大型的焊接工厂，一些生产车间都要做一些防静电。人体的静电达到几千伏是很正常的，因此我们就使用到了这 2 个电容，主要任务是来抵抗电磁干扰。

首先讲第一个电容，电容有很多种，下面讲几种常用的类型：

第一种，胆电容，它和黄豆大小，这个有深色条的是正极，另外一端表示负极，原理图上，它的正极接到了正 5V，负极直接接到了 GND。这种的电容特性比较好，它是贴片的，占用的面积也比较小,但是价格比较贵。如下图 2-40 所示。

第二种是电解电容，如下图 2-41，它是插件的。电解电容的个头非常大，它的容值也可以做得非常大。它的缺点，最大的缺点就是占的空间大，当然一些特性也比不上胆电容。但是它的价格非常便宜，因此日常用的非常多，像我们板子上用到的就是 1 个点解电容。可以看一下，在我们板子开关的旁边就是一个胆电容。那么这个电容呢，容值非常大，只是占用的空间大了一点，因此呢，像我们做开发的、学习来讲，用便宜的就可以了。



2-40 胆电容图



2-41 电解电容图

第三个是贴片和陶瓷电容，如下图 2-42，像我们这种贴片的都是陶瓷电容，特性来讲，陶瓷电容是最好的，但陶瓷电容想做到容值很大的话，它的成本就会很高，也不是太合适。因此像小个头并且容值比较小的用陶瓷电容比较多；容值大的用的就相对少一点。



图 2-42 贴片陶瓷电容

电容到底有什么作用呢？首先它的第一个作用，起到去除电源低频纹波，什么概念呢？大家看到大海，或者是河里的水，它这个水过来的时候，都是有那个波浪的。一起一伏一起一伏，那么对于我们的电流来说，它和这个水流是非常相似的。水来的时候是一起一伏一起一伏这样的过程。这个起伏过程如果直接加到我们后面的这个不管是单片机还是其他的一些电子器件，加到它们身上的时候，它会直接冲击我们的器件，水流起伏冲击到那个地上，长时间冲击的话，就会冲出个大坑来。其次，我们器件的这个工作电压，假设是 5V 的话，那么它这个有波浪的话，就会产生 4.8、4.9、5、4.7 伏，就会产生一个低频的纹波，也就是说它的电源是不稳定的，电压是不稳定的，这样对我们电子设备也有一定的影响。因此它在这里相当于一个水缸、水池的这样一个作用，缓冲前面来的这样一个水流。这个水流流经过这个水缸的时候变得平稳了，非常平稳的 5V 再提供我们后级的这个电子设备的使用。因此它在这里就起到一个水缸的作用。

第二个作用就是储水池的作用，为什么叫储水池呢？因为电的设备啊，我们的电设备，后级用电是在变化的，比如说我们的设备刚开始用的是 10mA 的电流，突然，某个设备变为 100mA 了。那么，如果没有一个储水池的话，大家可以想象到，水流流过来的时候，如果水流很小的话，突然后级遇到一个大坑，那么这个水位啪就降下来，也就是对于我们这个电流来说，它的电压突然降低了，直接影响到整个系统的软件系统。用到了这个储水池之后呢，

如果后级电流从 10mA 突然到了 100mA，那么我们这个储水池有足够的水提供给后端的电子设备使用。那么它在一定的程度上整个系统的电压更平稳一些，那么它在这里就起到一个稳定电源的作用，它主要是稳定电压的，那么这个电容是非常非常重要的。我们在设计电路的时候必然要用到它。

下面再详细举例说明一下电容的用法，下图是神舟 51 单片机开发板的 USB 接口电路原理图，如图 2-43 所示：

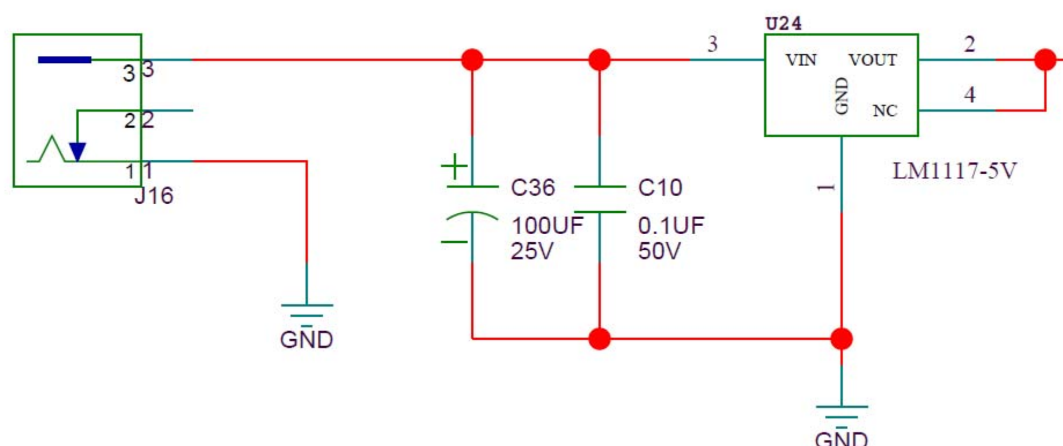


图 2-43 神舟 51 单片机开发板的 USB 接口电路原理图

首先电源的输出端这个电容是必须的，我们在设计 USB 电路的时候，进来时先一个储水池，它在这里起到一个稳定电源的作用。后边的所有电路，都通过这 2 个储水池稳定电源。那么一些供电比较大的系统，比如像彩色液晶屏，耗费电流比较大，再比如步进电机，步进电机耗电也比较大。因此我们在它们的旁边都要加这样的电容，。并且在电路布局的时候，一定要紧靠着这些器件，这样滤波的效果才会比较佳。

那么选取这个电容的指标有哪些呢？首先，像这个电容通常都有 2 个指标：

第一个指标是耐压值，就是它可以抵抗多少的电压，第二个指标是它的容量大小。首先看下耐压值，耐压值呢，它是 5V 的系统，那么它最少也是 5V 通常它想要安全工作，就要 2 倍的这个耐压，就是我们的整个系统是 5V，那么它本身耐压就要达到 10V 以上，这个是一个安全的情况。对于我们这个电路来讲，我们电容用到的是 16V，100uF，我们的耐压是用到 16V 的，因此呢，我们整个电路是 5V，用到 16V 的耐压是非常安全的。那么对于一些电压比较高的场合，这个电容耐压值稍微一大，这个个头就会变得很大，有时候在 1.5 倍左右也是可以的。但是呢，这个概念大家要清楚，尽量保持在整个电压系统以上，那么这个电容本身就是安全的。否则的话，这个电容容易坏掉，这个是第一个电容的作用，也是第一个指标。

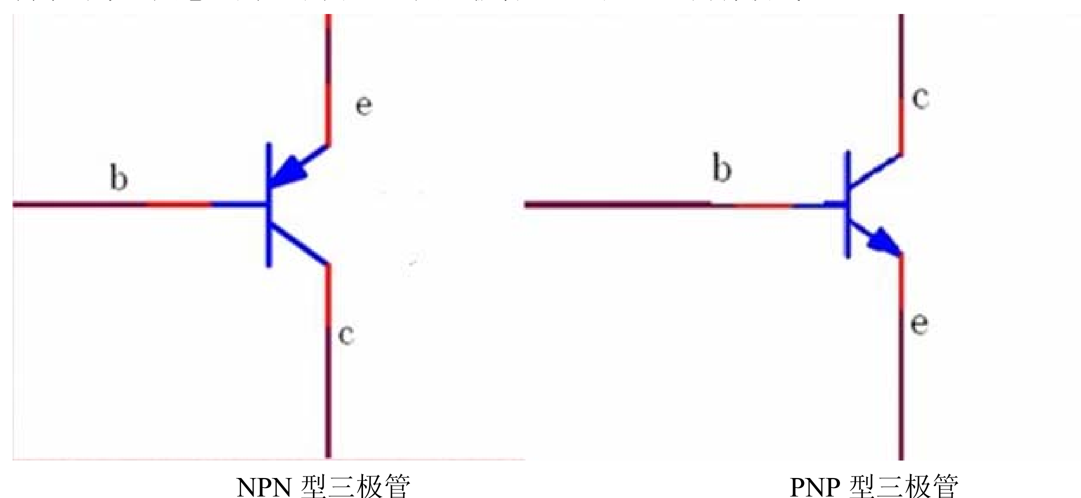
第二个指标是它的容值，我们这个上面用的是 100uF，像我们用到较多的是 100uF、470uF、1000uF、2200uF，这个电容的大小呢，完全是根据我们这个电路设置来决定的，就是我们电路设置耗电量越大，这个电容应该就越大，因为从 0 开始耗电量特别大的时候，它要提供足够的电流，提供给后级，前面应该储水池提供足够的电流给后级，因此它的大小完全是由电路来确定的。当然，像我们这个开发板耗电是非常低的，我们都是用 100uF。假如你在设置一款产品的时候，确定不了电容的容值的时候，就先把别人的电路拿过来看一下，学一学，比如你要设置一款开发板，你把我这个电路拿过去，学习一下，先搞明白这个电路

是怎么回事，然后就可以直接用，因为经过实践检验了的。

还有一个是 0.1uF 的电容做什么用途呢？。这个电容呢本身就比较简单了，它主要任务是过滤一些小的干扰，比如静电、快速瞬间脉冲等高频信号，我们知道，电容对这个高频信号相对应一个短路，因此这个电容相对应短路一些那些高频干扰的，相对于我们的手接触到这个电容产生的静电，经过这个电容短接到地消除了。这个电容就起到这样一个作用，这个电容的选取呢，就相对于简单一点。因为这个电容是根据前面大多前辈们的实验，他们根据这个电路的布法、电容的特性，根据这个所有电的一些设备的特性，总结出来，用这个电容是比较好的，0.1uF 就是我们平时用的 104,104 是 10 乘以 10 的 4 次方皮法。这个电容的应用是非常多的，几乎每个板子都用到。因为静电脉冲，空气静电。这个干扰都是非常多的，我们是感觉不到空气中的这个静电的，空气静电，是非常多的，它会产生一些干扰。这个电容就是去到消除这个干扰的作用，在整个电路板上是使用最多最多的一个器件。慢慢大家在看其他电路板的时候也会有一个这样的感触，这个电容用得最多。

2.9.3 三极管

这个三极管在控制蜂鸣器电路中起到了一个总开关的作用，详细的解释后面自然会懂，现在首先来认识一下三极管。三极管是我们日常生活中经常会用到的一个器件，也是非常非常重要的。分为 PNP 和 NPN 型，根据它的材料可以分为硅管和锗管，我们一般情况下，讲这个硅管就可以，锗管我们以后再慢慢去了解。我们主要讲硅管，从型号上来分 NPN 和 PNP。三极管在电路中，我们最常用的有三个功能，第一是开关控制；第二个的作用是信号放大，它有个放大功能；第三个是电平转换，这些都是比较实用的三个功能。从原理图上我们可以看到，三极管有一个箭头，有三个方向，伸出三条腿来，那么我们如何去判别它的型号，如何学习来它是怎么用，下图 2-44 是三极管 NPN 和 PNP 两种款式：



NPN 型三极管　　　　　　　　PNP 型三极管

图 2-44 三极管 NPN 和 PNP 两种款式

这是两个三极管，一个是 PNP 型，一个是 NPN 型。三极管首先要知道它有三极，有三个方向，其中红的这极就是 B 级也叫做基级，剩下的两个方向一个是发射极，一个叫集电极，大家可以先不用记，等会会记住这些东西。在认 PNP 型三极管和 NPN 型三极管的时候经常会搞混，不过我总结了一句话，以后只要记住这句话，三极管的型号就再也不会搞混了。首先 PNP 型和 NPN 型我们只看它最后的一个字母，先看下 PNP 的，它的这个 P 我们就假设它的一条腿，NPN 型呢，就是 N 的一条腿，那么我们从这里可以看出，它 P 的这条腿在

整个 PNP 的内侧，NPN 的那条腿在 NPN 上是朝外的。好了，那么规律就来了，箭朝哪就朝哪。这个是 PNP 型三极管，它的箭头是朝里的，而那个 P 是朝里的，这个 NPN 呢，它的箭头是朝外的，因此呢，这个 N 字母的那条腿是朝外的。从这个就能直接看出来，它是 PNP 型还是 NPN 型，我们通过图马上就能知道它的型号，或者是要说出它的型号。马上就能画出图来了，首先先把箭头确定了，是朝里还是朝外的。导通电压向箭头方向过，我们三极管的导通电压高于 0.7V 也就是这个箭头有 0.7V 的压降，也就是当箭头有 0.7V 的时候，三极管就导通了，那么刚才讲过，它有 B 级、C 级和 E 级三个级。导通电压顺箭头过，对于 PNP 型三极管来说，C 级比 B 级高 0.7V，这里就有电流了，对于 NPN 型三极管来说，B 级比 E 级高 0.7V 就有电流了，导通电压顺箭头过，那么导通电压有 0.7V 了。电压导通，电流控制，什么是电压导通，电流控制呢，电压导通的概念就是说，只要这个 PNP 型三极管 C 级比 E 级高 0.7V 就导通了，它导通是靠电压的，那么它的电流控制是怎么样控制呢，像我们三极管起放大作用的时候，它有一个倍数关系，我们在这里只起到一个开关的作用。开关作用就是三极管在饱和状态，像我们数字电路就 2 中状态，一种是截止，一种是饱和，放大我们先不讲，电压一导通，那么它就导通了，那么如何让它工作在饱和状态呢，我们只要保证这个 B 级电流 100 倍大于 C 级和 E 级之间的电流，它的电流流向是这样的，B 级的电流是 E 级的电流往 B 级流。

举例说明一下三极管控制蜂鸣器的用法，下图 2-45 是神舟 51 单片机开发板的蜂鸣器电路原理图：

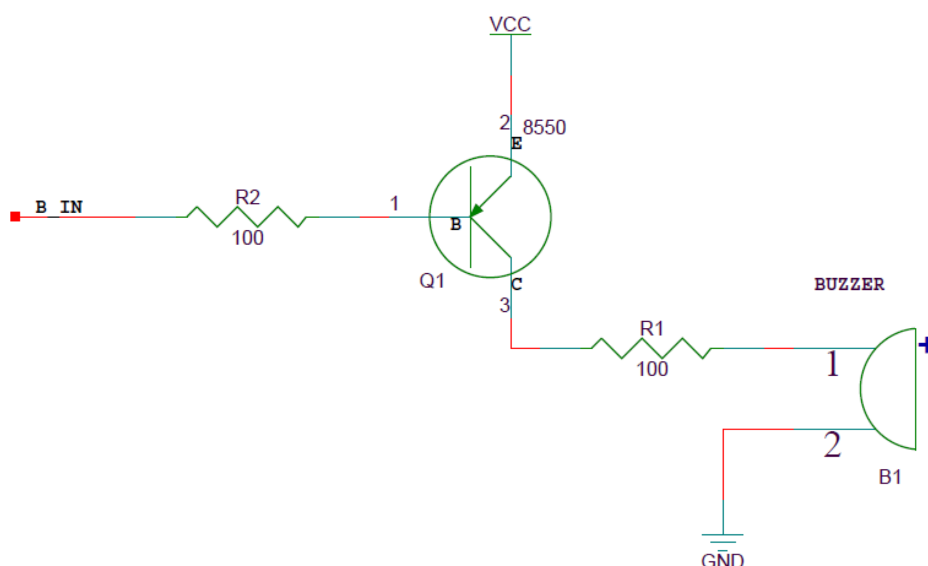


图 2-45 神舟 51 单片机开发板的蜂鸣器电路原理图

大家看一下这个，就是典型的例子，如果这个 I/O 口 B_IN 是单片机的 5V 系统（B 级），如果是高电平，三极管就导通了，一但导通，这条路就导通了，那么它就输出一个高电平了（C 级上的那根引线），如果 I/O 口输出一个低电平，三极管不导通，被截止，这里就输出一个低电平，也就是说用一个 5V 的系统控制了一个其他电压(VCC 可以为 5V，也可以 12V 或者其他电压值)的系统，用其他供电电压直接来驱动蜂鸣器响，而不需要用 IO 管脚来驱动 ，为单片机节约了功耗，达到了四两拨千斤的作用。

2.9.4 晶振电路

晶振可以说是 CPU 的心脏，晶振可以根据实际需要来选择是 6M、12M、11.0592M、20M

等等很多很多，这个晶振的接法是直接接到单片机的 2 个晶振引脚上。对地要接 2 个电容，这 2 个电容也叫负载电容，这 2 个电容是 10 到 30pF 之间都可以，这个是晶振厂家要求的，没有什么道理可以讲，你可以先不了解那么多，直接接 20pF 就可以，通用的，所有的单片机板子基本上都是适用的。假如单片机不工作，第一个万用表测电压，测试单片机的工作电压是不是工作在 3.8 到 5.5V 之间；第二个是测晶振，晶振通常是用示波器来测的，但也可以用万用表简易的测试一下，就是红表笔对晶振的引脚，黑表笔接 GND，测量它的电压，电压一般在 1 伏到 4 伏之间，如果是 0V 或者是 5V 的话，那么说明这个晶振没有正常工作。

2.9.5 复位电路

复位的功能说得更具体一点，就是单片机内部所有的寄存器初始化，像程序是 100 行，当运行到第 50 行的时候，突然掉电了，这时很多内容都是不确定的，那么在程序上电的时候，我们都会给它复位一下，进行一个初始化，51 单片机的复位时间大约在 2 个机器周期左右，具体需要看芯片数据手册，比如说我们用到的这个 89C51，它除了复位的时间外还需要让芯片稳定工作，所有说他要的时间长一点，这个时间超过 2 个机器周期就可以，但小于这个时间就不可以，怕没有复位时间不充分，导致单片机复位异常，具体的时间各位要学会阅读这个数据手册就可以，神舟 51 单片机的复位电路如下图 2-46 所示：

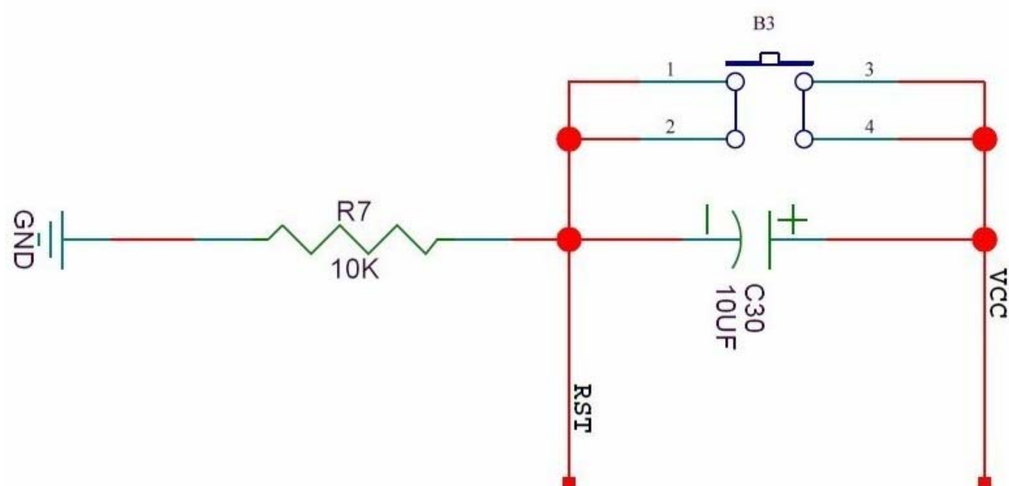


图 2-46 神舟 51 单片机开发板的复位电路

可以从原理图看到，神舟 51 单片机是高电平复位，低电平正常工作，我们用的是这样一个复位电路，这个电容在这里是起充放电的作用，下边是一个电阻，这个电阻就是一个下拉电阻，下面章节有讲解下拉电阻，大家看，这是一个按键，当按键没按下是松开状态的时候，电容对直流来说是隔离的，GND 和复位的这个接触点，这个电压不是 0 伏的话就有电流通过，实际上，上面有个电容给隔开了，因此上面的那个电压和下面的这个电压是一模一样的，是 GND。大家要记住，只要有电压差，只要这条电路是通的，那么它就有电流，这个地方它是不存在任何电压差的，因此在没按下按键的时候，它实际上是 0V。

之前讲了，是高电平复位，也就是当按下按键，这 5V 顺着这条支路流下来了，这个是 10K 的电阻，到了这个 RST 引脚的时候就是一个高电平，这个大家可以除一下，只要工作在这个 VCC 的范围之内就可以。当按下按键就是一个高电平，松开按键就是低电平，当它按下的一瞬间就是一个复位的状态，同样的道理，在上电的一瞬间，我们实际上是要给这个电容充电的，这个地方是 5V，我们电路在上电的时候，RST 引脚为低电平，连接到电容上，

当电容充满电的时候，实际上电容的电压等于 VCC 的电压，让单片机进行一个复位。

2.9.6 单片机IO口的状态

单片机有很多管脚，这些管脚负责控制外面的设备，比如下面图 2-47 这个电路图，用 VCC 驱动这个 LED 小灯，这个 LED 小灯就是发光二极管或者是数码管：

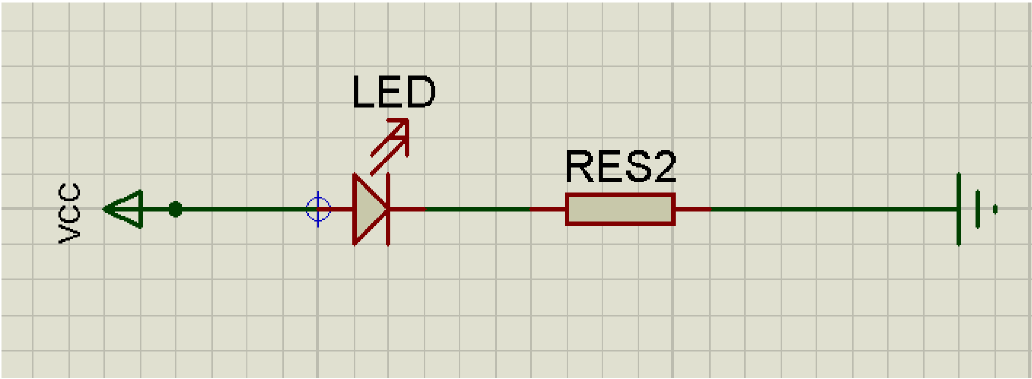


图 2-47 点亮一个发光二极管的电路图

现在当我们把 GND 去掉，换成 I/O 口，把 VCC 去掉，换成 I/O 口，在这里我们就能去了解，当 I/O 口输出大电流的时候，它是有一定的要求的，我们把单片机的这个 I/O 画出来了，我们看一下下图 2-48：

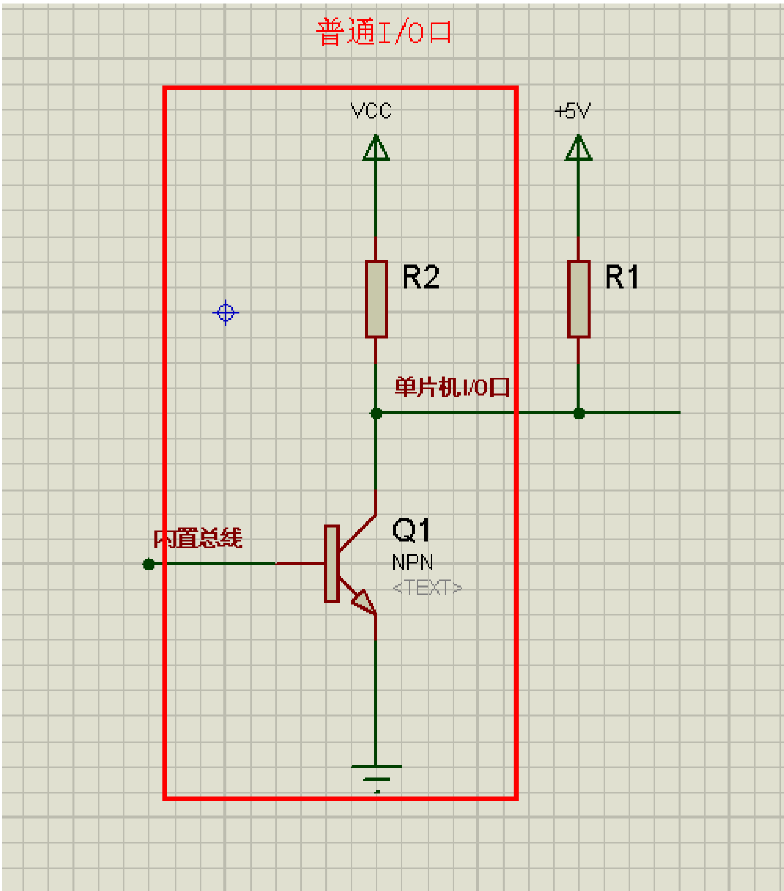


图 2-48 单片机管脚内部原理图

这个是单片机的一个 I/O 口，红色框里是单片机的内部，红色框外是单片机管脚外部，

内部和三极管的原理是一样的，所以我们用三极管来讲，这个是内部总线，就是单片机内部的一个框图，单片机这里输出一个高电平的时候，三极管不会导通，外部也输出一个高电平；当单片机输出一个低电平的时候，三极管导通，外部 I/O 口就输出一个低电平，在输出高电平的时候，这里有一个电阻，我们看一下手册，看一下 89C51RD 的手册，它内部也介绍了，比如 P0 口，P1 口，P2 口和 P3 口是什么类型的接口，这里下面会详细说。

下表 2-4 是 I/O 口工作类型的设定参数表：

表 2-4　I/O 口工作类型的设定参数

I/O 口工作类型设定		
P3 口设定<P3.7, P3.6, P3.5, P3.4, P3.3, P3.2, P3.1, P3.0>		
P3M0【7:0】	P3M1【7:0】	1/O 口模式
0	0	准双向口（传统 8051 I/O 口模式），灌电流可达 20mA，拉电流为 230uA，由于制造误差，实际为 250uA~160uA
0	1	推挽输出（强上拉输出，可达 20mA，尽量少用）
1	0	仅为输入（高阻）
1	1	开漏，内部上拉电阻断开，要外加
P2 口设定<P2.7, P2.6, P2.5, P2.4, P2.3, P2.2, P2.1, P2.0>		
P2M0【7:0】	P2M0【7:0】	1/O 口模式
0	0	准双向口（传统 8051 I/O 口模式），灌电流可达 20mA，拉电流为由于制造误差，实际为 250uA~160uA
0	1	推挽输出（强上拉输出，可达 20mA，尽量少用）
1	0	仅为输入（高阻）
1	1	开漏，内部上拉电阻断开，要外加
P1 口设定<P1.7, P1.6, P1.5, P1.4, P1.3, P1.2, P1.1, P1.0>		
P1M0【7:0】	P1M1【7:0】	I/O 口模式（P1.x 如做 A/D 使用，需先设置成开漏或高阻输入）
0	0	准双向口（传统 8051 I/O 口模式），灌电流可达 20mA，拉电流为 230uA，由于制造误差，实际为 250uA~160uA
0	1	推挽输出（强上拉输出，可达 20mA，尽量少用）
1	0	仅为输入（高阻），如果该 I/O 口需作为 A/D 使用，可选此模式
1	1	开漏，如果该 I/O 口需作为 A/D 使用，可选此模式

可以从图 2-49 中看到 I/O 口是准双向口，它是可设的，是可编程的；这就是说我们内部可以进行一个设置，另外这个特殊功能寄存器在使用的时候，如果不进行设置，它默认的是 00，也就是标准的准双向口，它的灌电流比较大，可以输入 20 毫安，它往外输出电流大概是 250 微安到 160 微安这个范围之内，但是大家要注意，单引脚灌电流达 20 毫安，但是整体电流推进不要超过 55 毫安，否则可能影响到单片机芯片的寿命；因此可以从这里看到，单片机是一个控制模块，控制中心，而不是一个驱动器。因此要避免给单片机输入或者是输出一个大电流，这个是它的普通 I/O 口要注意的；第二个是 01，现在我们先了解，后边我们使用的时候再说，最重要如何使用我们不用了解怎么设置，第二个是一个强推挽输出，可以达到 20 毫安，什么是推挽输出，先看下图 2-51，这个是一个普通的 I/O 口，是带上拉电阻的，这种是强推挽输出，我们看一下：

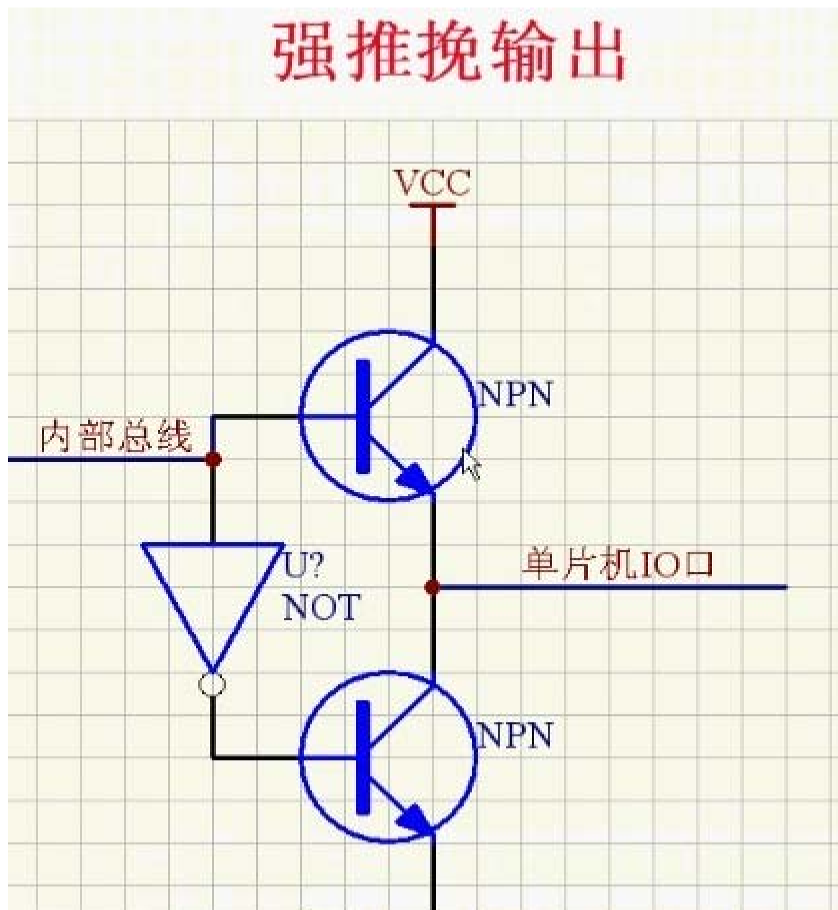


图 2-49 单片机管脚强推挽输出内部原理图

这是一个内部总线，它几乎是没这个上下拉电阻的，一但内部总线输出一个高电平，把三极管就导通了，导通后这个电流就下来了，这里是高电平，这是一个反相器，是一个低电平，下面这个三极管是截止是不导通的，电流就直接从上面的三极管直接输出了，没有限流电阻的话，这个电流是非常大因此呢，外部需要加限流电阻，具体需要注意，不接上拉电阻的话，看下数据手册吧，也有可能烧坏单片机的引脚。当低电平的时候，上面的三极管是戒截止的，一但产生低电平，经过反相器，第二个三极管接的是一个高电平，三极管导通，这个三极管输出电流也是非常大的，因此外部也要加限流电阻，这个是一个强推挽输出。

第三个是开漏，又叫开集，它内部总线经过一个反相器，原理图如下图 2-50 所示：

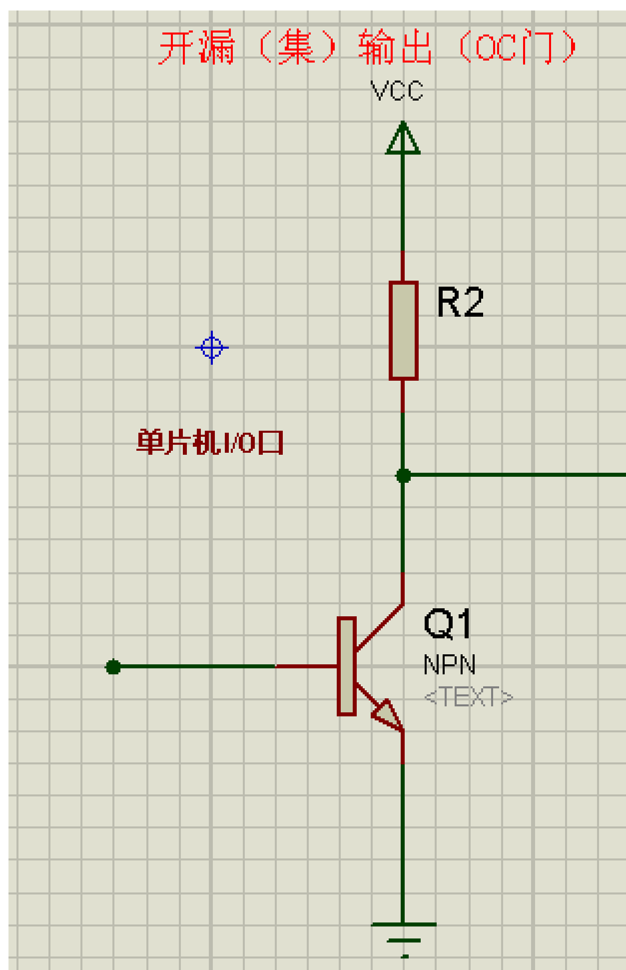


图 2-50 单片机管脚开漏输出内部原理图

假设是一个高电平的话，经过反相器变成低电平，低电平它内部是没有上拉电阻的，它内部是开着的，即使是高电平，导通三极管也不会输出一个高电平，低电平还是输出一个低电平，因此开漏的时候，外部一定要加上拉电阻，它才有可能输出高电平，不加上拉电阻的话它就不会输出高电平。比如说我们这个 89C51 的 P0 口就是一个开漏的双向 I/O 口，写"1"的时候，P0 口悬浮可用作高阻态输入，如果说你写 1 的话，外部要是没上拉电阻的话还是一个悬空的，不可能输出一个高电平，所以外部一定是要加一个上拉电阻的，它的作用也很多，比如在我们的实际运用系统中，我们也要避免直接接高电平，它可以在我们做电平转换的时候，就可以用这条电路，我们一个 5V 的系统要控制一个 12V 的系统时候，我们用 5V 的电平控制三极管的导通，一但导通了，外部就是 12V，那么可以进行一个电平的转换。在这里我们要避免单片机直接接，我们这里讲的只是一个电路，大家充分的用到这条电路，进行电平的转换。

2.9.7 上下拉电阻

上拉就是将不确定的信号通过一个电阻钳位（钳位就是保持的意思）在高电平，电阻同时起限流作用。下拉同理。也是将不确定的信号通过一个电阻钳位在低电平。

上拉是对器件输入电流，下拉是输出电流；强弱只是上拉电阻的阻值不同，没有什么严格区分；对于非集电极（或漏极）开路输出型电路（如普通门电路）提升电流和电压的能力是有限的，上拉电阻的功能主要是为集电极开路输出型电路输出电流通道。

可以想一下，不确定的信号就是我们不知道是高电平还是低电平，它加上一个电阻就是保持在高电平，另外它还起到一个限流的作用。比如说三极管，我们不知道单三极管到导通的时候，输出的是一个什么样的电平信号，这个电平信号和外部关系是很大的，但是一但我们加上这个上拉电阻之后，默认的三极管低电平不导通，一但这个三极管不导通，这个外部就会长时间保持在高电平，这个电阻就保持了高电平的作用。但是一但这个三极管导通了这里就会有电流通过三极管流下来，那么这个电阻又起到了一个限流的作用，所以上拉电阻其实有 2 个作用，一个作用呢，是嵌位高电平，另外一个作用呢，就是限流。实际上三极管不导通的时候，直接接 12V 是可以的，但是当三极管一但导通的时候，这里没有电阻就不行了，所以起到了嵌位、限流的作用。下拉同样的道理。

上下拉电阻的主要作用非常多，它的应用在这 4 个电路中用得比较多：

第一个是电平转换的，它可以输出电平的数值，刚才讲过了，用 5V 接三极管达到控制 12V 的作用，加了一个上拉电阻。

第二个是 OC 门，OC 门刚才我们讲过，P0 口是一个开漏的，三极管不管是否导通，都不能输出一个高电平，因此开漏输出必须需要加上拉电阻。标准的 51 的 P0 口是一个开漏输出，89C52 其实已经给你设置了，开漏输出的 I/O 口你想要它输出一个高电平必须加一个上拉电阻才能正常使用。

第三个是加大普通 I/O 口的驱动能力，我们前面也提到过了，普通 I/O 口的输出能力也就二百多个微安，如果电流比较大的话，这里加上一个上拉电阻，那么 I/O 口的输出能力就是 2 个上拉电阻的和，加这个上拉电阻起到一个增大驱动能力的作用。

第四个功能是悬空引脚加上下拉抗干扰。就是有些悬空的引脚大家避免单片机引脚的悬空，其实悬空也没事，尽量避免，悬空的话，在一定程度上有非常非常细微的功耗在里边，而且容易受干扰，比如空间放电，静电等等这些干扰，如果说你有这个上下拉电阻的话，嵌位，让它长期保持在高电平或者是低电平。那么它就会避免这个问题的出现，因此悬空的引脚在设置的时候，尽可能的让它上下拉。当然了，一下简单的小系统不接也没关系，那么在使用上下拉的时候，我们要注意什么问题，就是这个电阻的大小如何选呢。

选取上下拉电阻值也有讲究，具体经验总结如下：

1. 从节约功耗及芯片灌电流能力考虑应当足够大，因为电阻大，电流小，可以节约功耗，还有这个灌电流，单片机输入电流要尽可能小，对单片机影响小一点，延长单片机使用寿命。

2. 从确保足够的驱动电流考虑应当足够小；电阻小，电流大，比如说你驱动应该发光二极管，如果你这个电阻小了的话，这个发光二极管就不亮了，所以从确保这条路的情况下，这个电阻应该小一点。这个 1 和 2 似乎是矛盾，实际取多大的电阻完全是根据你的应用这个设计。

3. 对于高速电路，过大的上拉电阻可能会导致边沿变平缓，这个是什么概念呢，因为我们是数字电路，数字电路从 0 变成 1 它实际上是一个缓慢的一个上升过程。并不是说 0 啪 90 度角直接上去，通过一个斜坡，如果说你这个上拉电阻过大的话，会使得你这个斜坡变得平缓，所以综合考虑：上拉电阻常用值在 1K 到 10K 之间选取，下拉同理。当然，在一些场合上百 K 也是有可能的，刚学的大多情况下都是 1K 到 10K。

第三篇 51 单片机全方位实战篇

3.1.如何下载第一个程序到单片机里

3.1.1 什么是冷启动

每个芯片都有自己的下载方式，目前下载方式比较常见的一般是利用 CPU 的串口进行下载，或者在 ARM 的 CPU 中大部分都有 JTAG 或者 SWD 的下载方式。

在这里，51 单片机尤其是 89S51 系列的一般都是用 CPU 的串口进行下载，当然这里首先要扫盲一些基本概念，就是关于什么是单片机的冷启动和热启动的问题？

1）冷启动——是指在断电状态下重新上电。冷启动，是在下载程序开始时，为了是单片检测有无下载信号。若有则下载；若无则执行原来的程序。

2）热启动——是指已经处于上电状态，给复位端加复位信号（还有其他类型的复位），程序重新运行。

神舟 51 的单片机是来自 STC 公司的，这颗芯片是属于冷启动的，所以在每次下载程序的时候，需要先按单片机开发板的开关，使其断电之后再上电来完成一次下载。

3.1.2 环境搭建

1. 打开已下载的本书配套资料，选择“步骤 3 安装 USB 驱动程序\xp 驱动\SETUP\USB_Driver.exe”安装驱动(根据自己电脑系统选择 XP 或者 WIN7)。如图 3-1 所示：

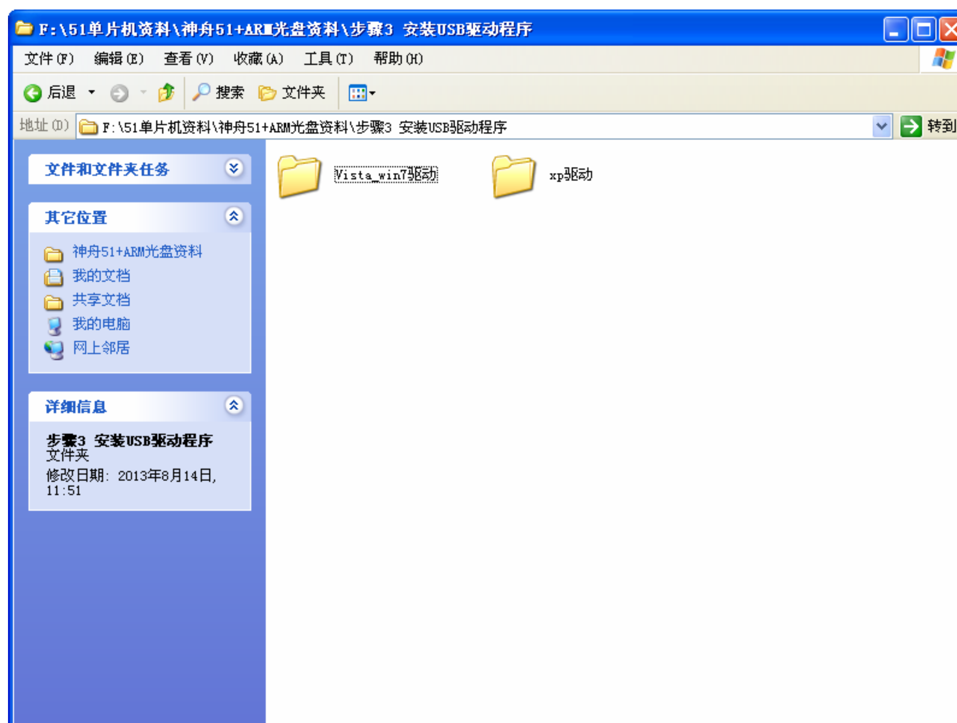


图 3-1 选择合适驱动文件安装

2.安装 MDK 的 51 编程开发工具，然后打开本书配套资料，选择“**步骤 4 各种配套软件\51 开发工具\KIEL C51 v7.5 工具软件\安装说明.txt**”安装驱动（**根据自己电脑系统选择 XP 或者 WIN7**），具体安装步骤可以看安装说明，如图 3-2 所示。



图 3-2 安装 KEIL 软件

3.1.3 开始下载第一个程序

1.在开发板上安装好芯片与下载模式的设置，如下图 3-3 所示：

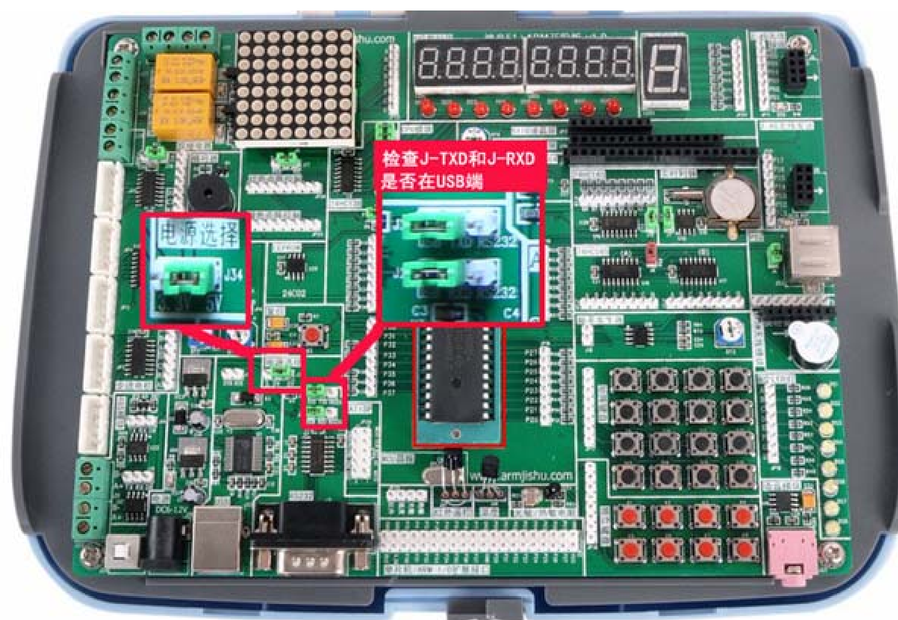


图 3-3 安装 51 单片机芯片与跳帽的设定

2.一根 USB 线负责通信和下载，一端连开发板，另外一端连 PC 端的 USB 口，如图 3-4、3-5：

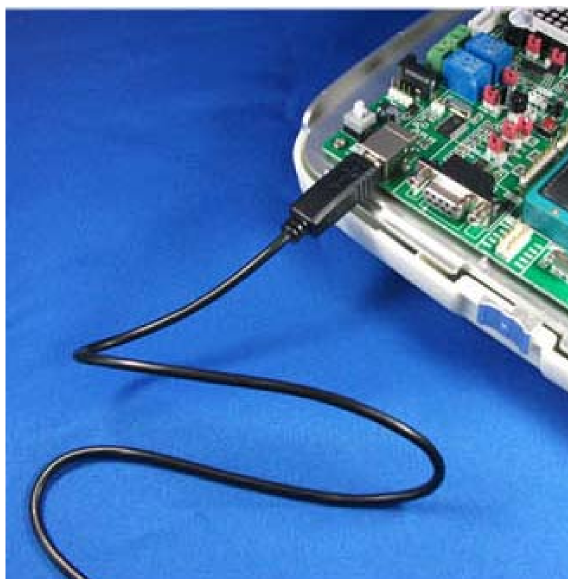


图 3-4 USB 线的连接通信

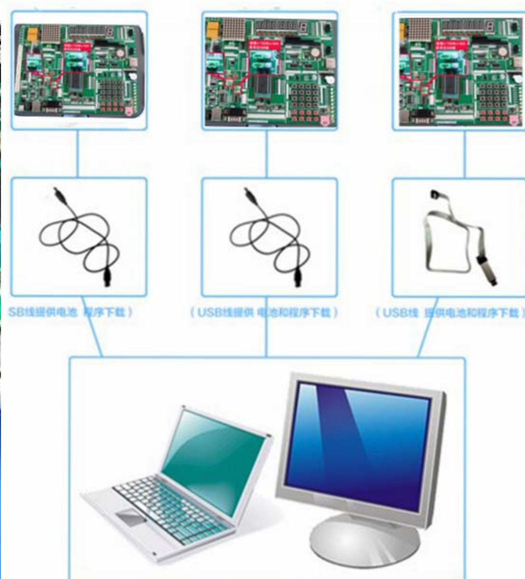


图 3-5 USB 线的连接通信

3. 单击“我的电脑”右键选择“设备管理器”，查看设备连接的端口，如下图 3-6 所示：

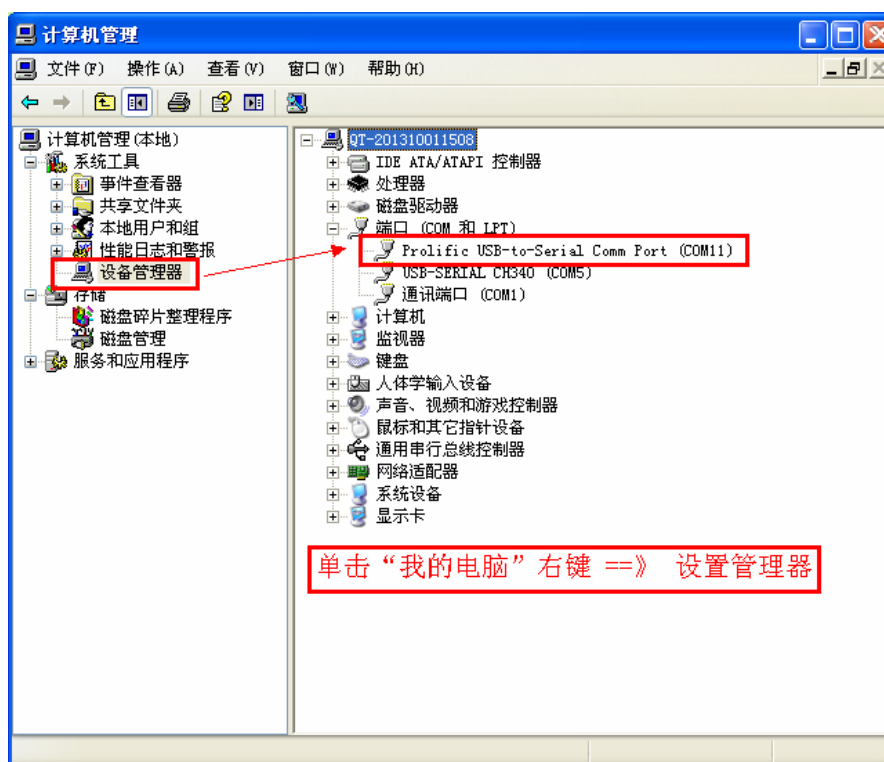


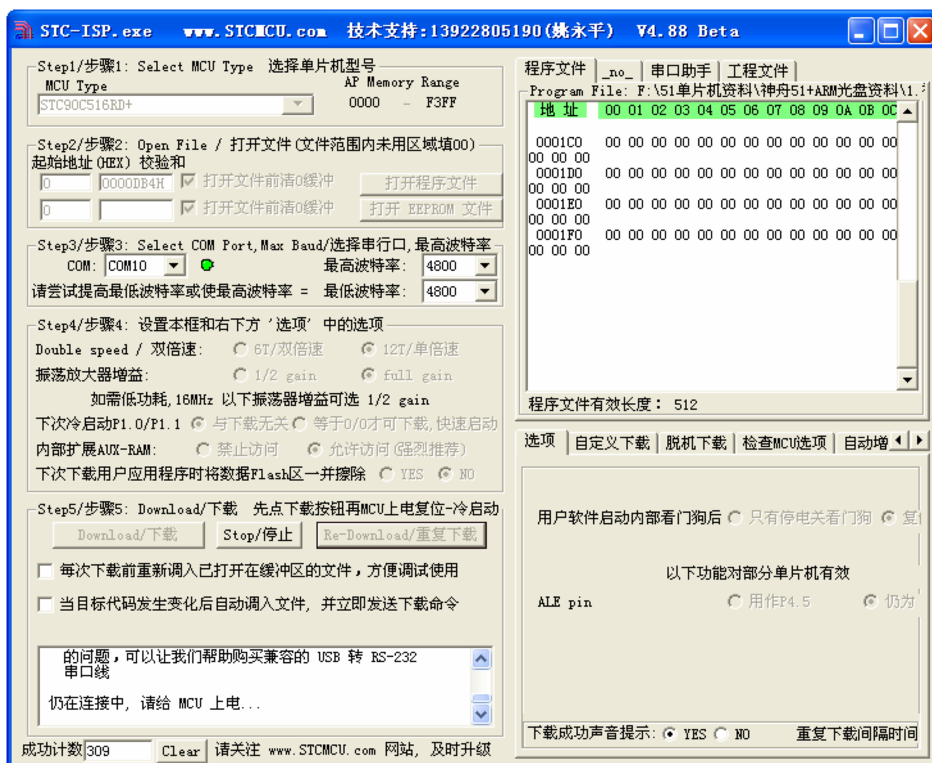
图 3-6　查看连接端口

4. 打开程序烧录软件，具体步骤如下图 3-7：



图 3-7 STC 官方下载软件的操作步骤

5. 完成之后，按下 Download/下载 按钮，如图 3-8 所示，使得这个软件在等待冷启动：



3-8 窗口提示信息

可以看到提示”仍在连接中，请给 MCU 上电…”

6. 完成之后，按下神舟 51 单片机开发板上的开关，先关电 1 秒钟，然后再上电，就会出现如下图 3-9 的下载提示：

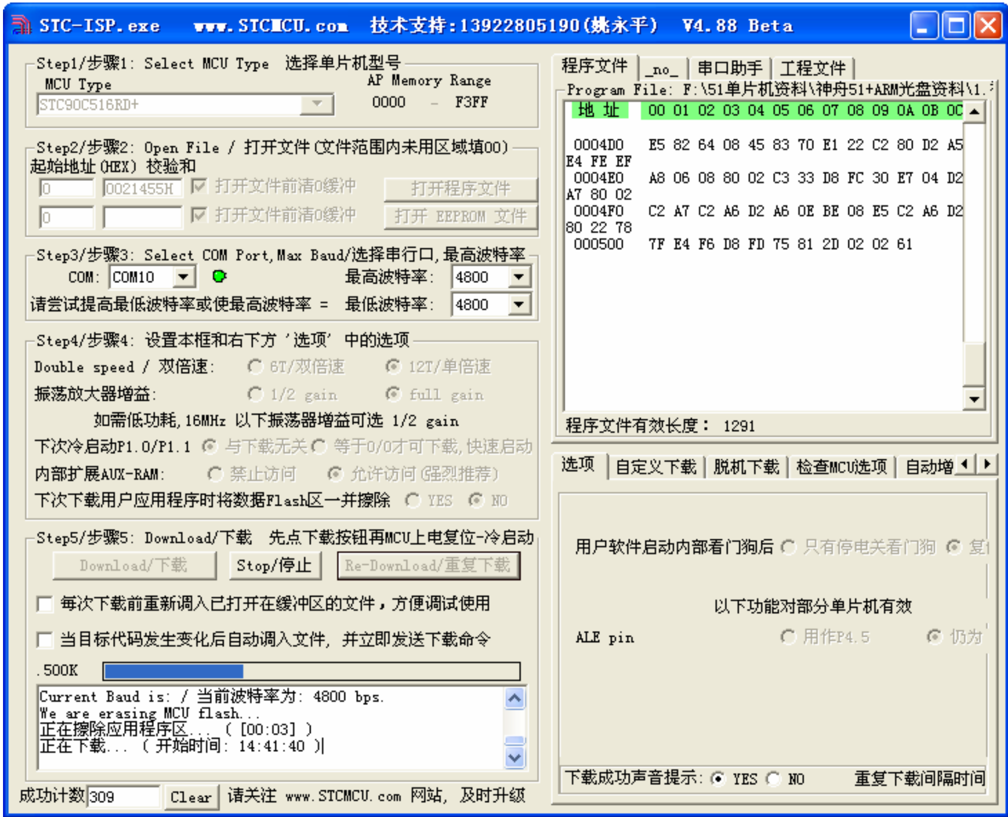


图 3-9 程序下载中

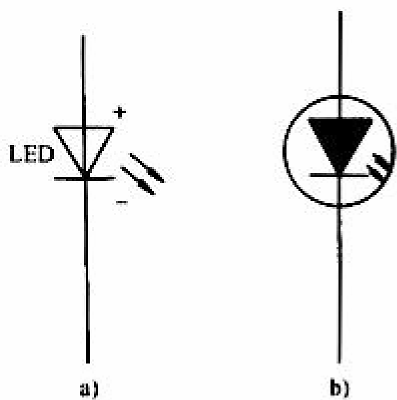
7.下载成功之后，重新上电，程序自动开始运行

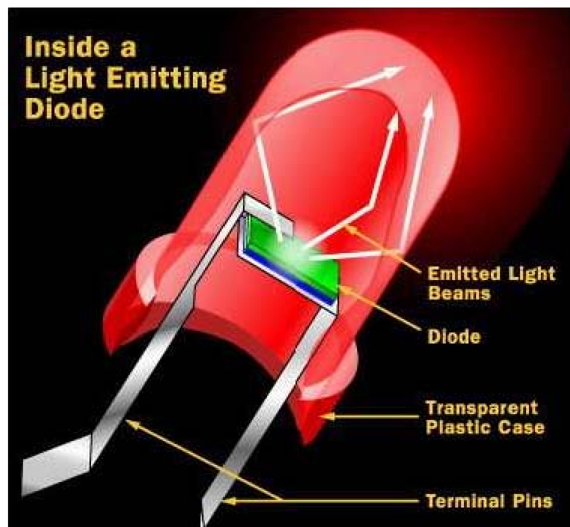
3.2 如何驱动发光二极管

3.2.1 发光二极管的原理

前面 51 单片机章节有简单描述发光二极管的使用以及原理，这里进行更进一步的深入探讨。

发光二极管（英语：Light-Emitting Diode，简称 LED）由镓（Ga）与砷（AS）、磷（P）的化合物制成的二极，它是半导体二极管的一种，是一种将电能转化成光能的产品，通过正负极加在某种物质两端，使得该物质产生发光的效果。符号与实物图如下图 3-10 所示：





3-10 LED 的符号与实物图

发光二极管加上正向电压，它就会产生自发辐射的荧光，因为发光二极管与普通二极管一样是由一个PN结组成，所以具有单向导电性（LED只能往一个方向导通）；不同的半导体材料中电子和空穴所处的能量状态不同，那么发光效果就不同，当电子和空穴复合时释放出的能量越多，发光效果就越强。

普通发光二极管的正向饱和压降为 1.4～2.1V,正向工作电流为 5～20mA;电流过大时会损坏 LED，因此使用时必须串联限流电阻以控制通过管子的电流。限流电阻 R 可用下公式计算，公式中 Vcc 为电源电压，U(led)为 LED 的正向压降，I 为 LED 的一般工作电流：

R＝（Vcc－U(led)）/ I

LED广泛应用于各种电子电路、家电、仪表等设备中、做电源或电平指示；发光二极管相对其他的灯源相比，工作电压更低（有的仅一点几伏）；工作电流很小（有的仅零点几毫安即可发光）；因为没有灯丝会烧坏，所以寿命就更长；此外，发光二极管外面是塑料比较抗震，使得更加持久耐用；传统的白炽灯发光过程产生了大量热量，这是对能源的一种浪费，因为这些热量并没有转化成有效电流去发光，而发光二极管所发出的热非常少，相对来说就是发光的性价比更高。

3.2.2 发光二极管的深入剖析

本章节涉及到了高中或大学的物理知识，对本节如果不明白并不影响后面的学习

二极管会发光的原因到底是什么呢？追其本源，假设某个二极管内部导体材料是铝砷化镓，对这种材料进行参杂和制作之后，材料比例不同，就会形成两种材料：一种是N型半导体，一种是P型半导体；N型半导体带有许多自由电子，向P型半导体的空穴，在对N型和P型半导体两端通电，就好比是发光二极管的两端通电，源源不断的为二极管两端形成电子流动，电子从N型往P型的空穴不断运动，从而不断释放出能量来。

光是能量的一种形式，而光子是光的最基本单位，是一种粒子；在整个电子从N型往P型的空穴的运动过程中，光子就被释放出来；能量以光子形式释放出来，被通电的发光二极管就会源源不断的释放出光子来。不同颜色的发光二极管通过过滤不同波长的光，就可以使肉眼看到不同的颜色。

3.2.3 硬件原理图连接

神舟 51 单片机 LED 部分原理图如下图 3-11 所示：

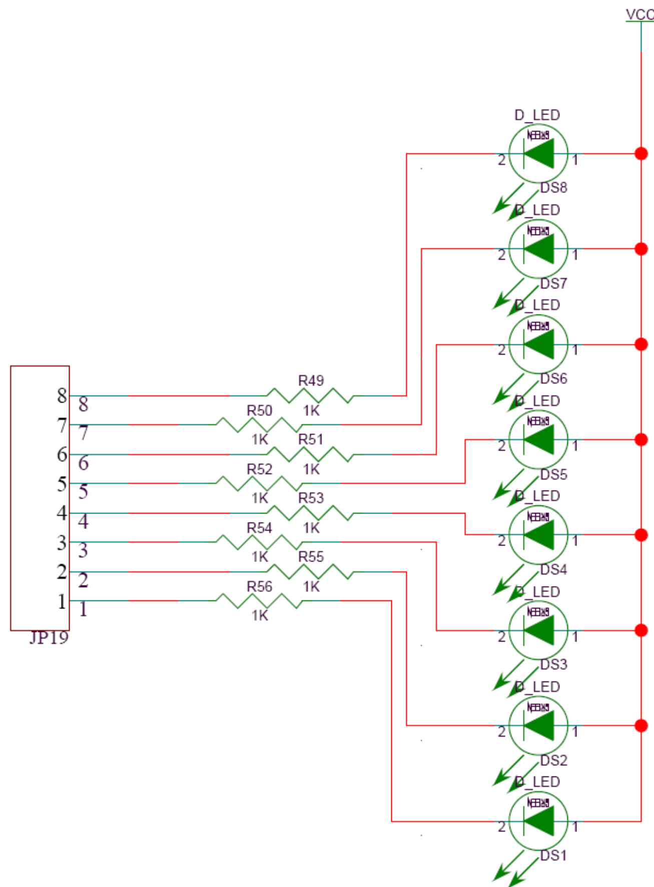
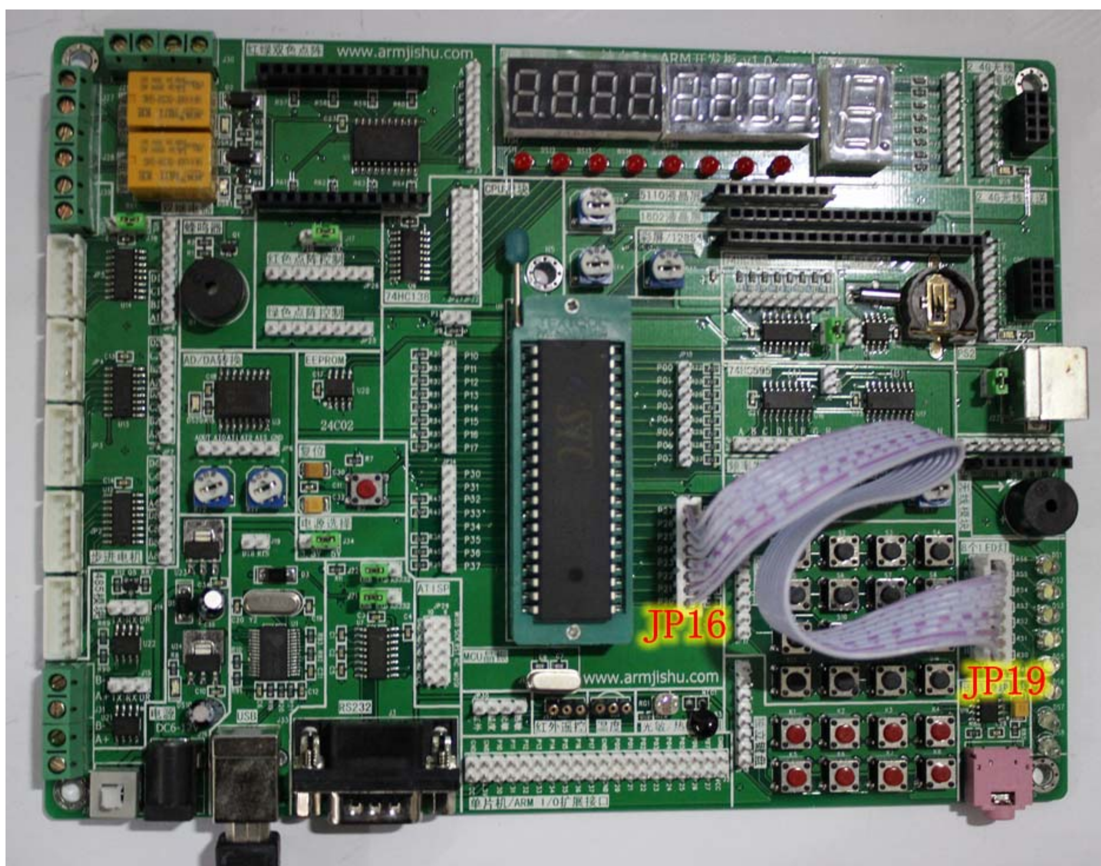


图 3-11 LED 原理图

从R49~R56总共8颗0603型号的1K（1K=1000）欧姆电阻，板子上的JP19 插针用于连接需要使用的IO口上。LED所有的样例中，该插针连接到单片机的P2口(排线连接到JP16)，如下图3-12：



3-12 LED 实验硬件连接图

连接关系如表 3-1：

表 3-1 硬件连接对应表

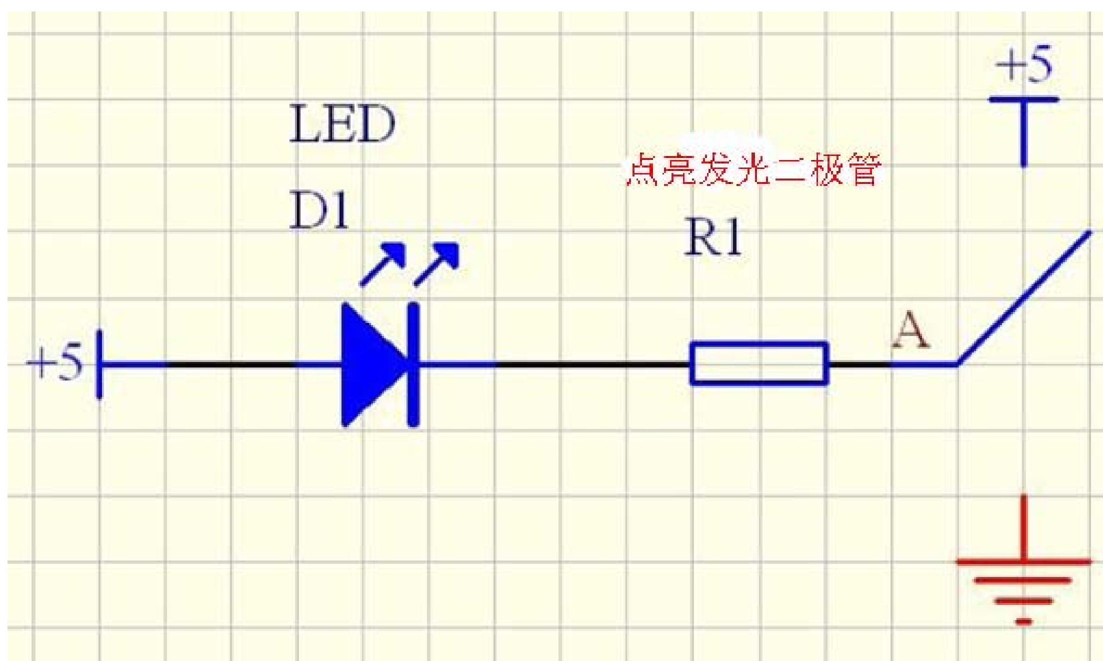
单片机接口	插座 1	方式	插座 2	线缆	功能
P2 口	JP16(A 向左)	直连	JP19（A 向右）	1 根 8 针扁平电缆	控制 LED 灯
实验现象：JP19 端低电平时 LED 点亮，高电平时 LED 熄灭。					

此实验之前首先需要了解IO端口原理，简单介绍如下：

1．I/O表示Input、Output，即输入输出。经常听到单片机端口是标准双向口，就是说，单片机的端口既可以作为输出信号端（如控制灯亮灭、继电器吸合释放、驱动电机转动等），也可以作为输入信号端（如按键信号输入、红外波形输入、无线信号输入，数据采集等）。

2．暂且不管IO内部详细结构，先把单片机当黑匣子处理，假设我们需要控制一个LED的亮灭，应该怎么做？

这里列出一个最简单的硬件控制电路，如图3-13：



3-13 LED硬件控制电路

上图中A点就相当于单片机的I/O口，A点相当于单刀双掷开关，可以接到+5V，也可以接到电源地。接到+5V或者悬空，整个电路中没有电流流过，LED的的状态是熄灭。如果A点接到电源地，2端压差5V，假设LED正常工作压降1.3V，正常工作电流3mA。我们通过图上的参数得知实际工作：I=（5V-1.3V）/1000欧姆=3.7mA，接近于正常工作电流，所以LED被点亮。在数字电路中，我们接+5V认为是电平“1”，接地就认为电平为“0”。

所以在单片机中，按照上图连接LED到P2.0口，我们只需要控制P2.0口的电平是“0”或“1”就可以控制LED的亮灭。

3.2.4 例程 01 单片机IO输出-点亮 1 个LED灯方法 1

程序代码如下：

```
/******************************************************
* 例程：IO 口高低电平控制点亮一个 LED 灯
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：点亮 P2 口的一个 LED 灯
        该程序是单片机学习中最简单最基础的，
        通过程序了解如何控制端口的高低电平
******************************************************/
#include<reg52.h>     //包含头文件，一般情况不需要改动，
                      //头文件包含特殊功能寄存器的定义

sbit LED=P2^0;   // 用 sbit 关键字 定义 LED 到 P2.0 端口，
                 //LED 是自己任意定义且容易记忆的符号
/*----------------------------------------------
                主函数
```

```
------------------------------------------------*/
void main (void)
{
                    //此方法使用 bit 位对单个端口赋值
    LED=0;          //将 P2.0 口赋值 0，对外输出低电平，LED 灯导通变亮
}
```

连接关系如表 3-2：

表 3-2 硬件连接对应表

单片机接口	插座 1	方式	插座 2	线缆	功能
P2 口	JP16(A 向左)	直连	JP19（A 向右）	1 根 8 针扁平电缆	控制 LED 灯
实验现象：下载程序后，可以看到由 P2.0 管脚连接的 LED 灯（DS8）点亮,其他的灯熄灭状态					

知识要点：

1、#include<reg52.h>，# 说明这是个预处理命令(在编译之前进行的处理)，include 是文件包含命令，提示在编译的时候预先包含 reg52.h 这个文件，C 语言的预处理主要有三个方面的内容：

1-宏定义

2-文件包含

3-条件编译

这里包含的 reg52.h 基本内容如下：

```
/*--------------------------------------------------------------------------
-
REG52.H
Header file for generic 80C52 and 80C32 microcontroller.
Copyright (c) 1988-2002 Keil Elektronik GmbH and Keil Software, Inc.
All rights reserved.
--------------------------------------------------------------------------*
/
#ifndef __REG52_H__
#define __REG52_H__
/*  BYTE Registers  */
sfr P0    = 0x80;
sfr P1    = 0x90;
sfr P2    = 0xA0;
sfr P3    = 0xB0;
sfr PSW   = 0xD0;
sfr ACC   = 0xE0;
sfr B     = 0xF0;
sfr SP    = 0x81;
sfr DPL   = 0x82;
sfr DPH   = 0x83;
sfr PCON  = 0x87;
sfr TCON  = 0x88;
sfr TMOD  = 0x89;
```

```
sfr TL0   = 0x8A;
sfr TL1   = 0x8B;
sfr TH0   = 0x8C;
sfr TH1   = 0x8D;
sfr IE    = 0xA8;
sfr IP    = 0xB8;
sfr SCON  = 0x98;
sfr SBUF  = 0x99;
/*  8052 Extensions  */
sfr T2CON  = 0xC8;
sfr RCAP2L = 0xCA;
sfr RCAP2H = 0xCB;
sfr TL2    = 0xCC;
sfr TH2    = 0xCD;
/*  BIT Registers  */
/*  PSW  */
sbit CY    = PSW^7;
sbit AC    = PSW^6;
sbit F0    = PSW^5;
sbit RS1   = PSW^4;
sbit RS0   = PSW^3;
sbit OV    = PSW^2;
sbit P     = PSW^0; //8052 only
/*  TCON  */
sbit TF1   = TCON^7;
sbit TR1   = TCON^6;
sbit TF0   = TCON^5;
sbit TR0   = TCON^4;
sbit IE1   = TCON^3;
sbit IT1   = TCON^2;
sbit IE0   = TCON^1;
sbit IT0   = TCON^0;
/*  IE  */
sbit EA    = IE^7;
sbit ET2   = IE^5; //8052 only
sbit ES    = IE^4;
sbit ET1   = IE^3;
sbit EX1   = IE^2;
sbit ET0   = IE^1;
sbit EX0   = IE^0;
/*  IP  */
sbit PT2   = IP^5;
sbit PS    = IP^4;
sbit PT1   = IP^3;
```

```
sbit PX1   = IP^2;
sbit PT0   = IP^1;
sbit PX0   = IP^0;
/*  P3  */
sbit RD    = P3^7;
sbit WR    = P3^6;
sbit T1    = P3^5;
sbit T0    = P3^4;
sbit INT1  = P3^3;
sbit INT0  = P3^2;
sbit TXD   = P3^1;
sbit RXD   = P3^0;
/*  SCON  */
sbit SM0   = SCON^7;
sbit SM1   = SCON^6;
sbit SM2   = SCON^5;
sbit REN   = SCON^4;
sbit TB8   = SCON^3;
sbit RB8   = SCON^2;
sbit TI    = SCON^1;
sbit RI    = SCON^0;
/*  P1  */
sbit T2EX  = P1^1; // 8052 only
sbit T2    = P1^0; // 8052 only
/*  T2CON  */
sbit TF2    = T2CON^7;
sbit EXF2   = T2CON^6;
sbit RCLK   = T2CON^5;
sbit TCLK   = T2CON^4;
sbit EXEN2  = T2CON^3;
sbit TR2    = T2CON^2;
sbit C_T2   = T2CON^1;
sbit CP_RL2 = T2CON^0;
#endif
```

1、从 reg52.h 的内容可以看到，头文件主要定义了端口和特殊功能寄存器的物理地址，包含这个头文件后，我们在程序中就可以直接使用定义过的标识符。如需要对 P2 进行操作，P2 的寄存器地址是 0xA0，我们不需要了解单片机具体内部结构和地址，直接针对 P2 进行操作，P2=0xFF；语句的作用是直接赋值十六进制 FF 到 P2 端口。

2、sbit 这个关键字是 C51 中特有的，用于定义 SFR(特殊功能寄存器)的位变量（什么叫位？一个字节有 8 个位变量）

例如：sbit LED=P2^0; 表示定义发光管连接的硬件端口，LED 定义在 P2（特殊功能寄存器）的第 0 位，即 P2.0，定义了这个端口以后，下面对 P2.0 的操作，我们就可以直接用 LED 代替，

```
LED=0;          //将 P2.0 口赋值 0，对外输出低电平
```

在数字电路的世界中，所有电信号都是由 0 和 1 来描述的，所以由于 sbit 定义位变量，赋值结果不是 0 就是 1。

3、main()函数：一个程序有且只有一个 main 函数，main()称之为主函数，是所有程序运行的入口。C 程序最大的特点就是所有的程序都是用函数来装配的；函数分为带参数的函数或不带参数的函数两种，参数均在调用的该函数的时候进行参数值的传递。

我们例程里的 main(void)加了个“void”表示 main 函数里没有任何参数，不加 void 也可以，void 表示空型。

本程序短小，但包含了一个 c 程序最基础的部分，以后的程序会在这个基础上不断添加内容以增加功能，整体结构不会改变。

4、 // 和 /* */ 这 2 种符号表示注释。

注释不是程序，不影响程序结果，注释是给我们程序员看的，我们可以通过注释了解程序的意图，尤其在程序庞大时，注释尤为重要，如果没有注释，一段时间后，我们自己写的程序自己都看不懂了。所以养成一个好的习惯，写程序是及时注释。

上述 2 种注释符号的区别如下:

// 后面的语句都为注释，换行后无效;

/* */ 中间的内容皆为注释，换行有效。

上述样例中开头的描述使用了/* */注释，而程序中各个语句后面的注释使用了 // 。这个可以根据个人喜好，没有具体要求。

3.2.5 例程 02 单片机IO输出-点亮 1 个LED灯方法 2

代码如下：

```
/******************************************************
* 例程：IO 口高低电平控制点亮一个 LED 灯
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：点亮 P2 口的一个 LED 灯
        该程序是单片机学习中最简单最基础的，
        通过程序了解如何控制端口的高低电平
******************************************************/
#include<reg52.h>      //包含头文件，一般情况不需要改动，
                       //头文件包含特殊功能寄存器的定义
sbit LED=P2^0;  // 用 sbit 关键字 定义 LED 到 P2.0 端口，
                //LED 是自己任意定义且容易记忆的符号
/*------------------------------------------------
                主函数
------------------------------------------------*/
void main (void)
{
                       //此方法使用 bit 位对单个端口赋值
    LED=1;             //将 P2.0 口赋值 1，对外输出高电平，LED 灯不亮
    LED=0;             //将 P2.0 口赋值 0，对外输出低电平，LED 灯导通变亮

    while (1)       //主循环 (while() 是单片机的一种基本循环模式。
```

```
                    //当满足条件时进入循环，不满足跳出)
        {
                    //主循环中什么都不执行，一直死循环，
                    //程序一直运行，P2.0 口所连 LED 灯一直长亮
        }
}
```

连接关系如表 3-3：

表 3-3 硬件连接对应表

单片机接口	插座 1	方式	插座 2	线缆	功能
P2 口	JP16(A 向左)	直连	JP19（A 向右）	1 根 8 针扁平电缆	控制 LED 灯
实验现象：下载程序后，可以看到由 P2.0 管脚连接的 LED 灯（DS8）点亮,其他的灯熄灭状态					

知识要点：

1. while()：计算机的一种基本循环模式。当满足条件时进入循环，不满足跳出。C 语言中 while 可以有以下 2 种形式：

形式 1：　　do <语句> while(<条件>);

形式 2：　　while(<条件>) <语句>;

我们分析一下本例中的意义，while(1){} 小括号里是条件，大括号中是满足条件后应该执行的语句。

当条件为真，执行大括号中的语句，否则跳出循环。条件不为 0 即为真，所以这里我们使用数字 1，表示条件永远为真，循环就不会停的执行，程序一直在运行，除非断电或者复位才会重新从 main 开始执行，当 while 为真的时候就会一直执行大括号中的语句，本例中不需要其他操作，所以大括号中没有写代码；

以后的所以程序中，都会存在 while 循环主题，所以需要循环操作的函数和变量都需要放在 while 循环中。本例中进入 main 函数对 LED 进行了两次此操作后进入 while 循环，并一直在循环中等待，不进行任何操作。

3.2.6 例程 03 单片机IO输出-点亮 1 个LED灯方法 3

代码如下：

```
/*********************************************************

* 例程：IO 口高低电平控制点亮一个 LED 灯
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：点亮 P2 口的一个 LED 灯
        该程序是单片机学习中最简单最基础的，
        通过程序了解如何控制端口的高低电平
*********************************************************/
#include<reg52.h> //包含头文件，头文件包含特殊功能寄存器的定义

/*------------------------------------------------
                    主函数
------------------------------------------------*/
void main (void)
```

```
{
  /* 此方法使用 1 个字节(即 8 个 bit 位)对单个端口赋值（P2 端口是由 P2.0~P2.7 总共
8 个位组成）*/
    P2 = 0xff;        //P2 口全部为高电平，对应的 LED 灯全灭掉
                      //16 进制 0xff 换算成二进制是 1111 1111
    P2 = 0xfe;        //P2 口的最低位点亮，可以更改数值是其他的灯点亮
                      //0xfe 是 16 进制，0x 开头表示 16 进制数，
                      //fe 换算成二进制是 1111 1110
    while (1)         //主循环
    {
                     //主循环中添加其他需要一直工作的程序
    }
}
```

连接关系如表 3-4：

表 3-4 硬件连接对应表

单片机接口	插座 1	方式	插座 2	线缆	功能
P2 口	JP16(A 向左)	直连	JP19（A 向右）	1 根 8 针扁平电缆	控制 LED 灯
实验现象：下载程序后，可以看到由 P2.0 管脚连接的 LED 灯（DS8）点亮,其他的灯熄灭状态					

知识要点：

与第一个程序不同，这里更简单，直接对 P2 口整体操作，我们使用的是 8 位单片机，所以一个口占的宽度是 8 位，用二进制表示 xxxx xxxx ，样例注释有详细解释。

0xFF 和 0xff 相同，c 语言中数值不区分大小写，标识符一定要区分大小写。0x 前缀表示这个是十六进制，如果直接用十进制则可以这样表示：P2=255 等效于 P2=0xff，单片机中常用的进制是 十进制、二进制和十六进制，三种进制只是表现形式不同，可以相互转换。

在写程序之前需要详细了解这 3 中数值并能熟练进行换算。最简单的办法可以通过电脑自带的计算器。这里演示一下：

打开 开始->程序->附件->计算器，如下图 3-14

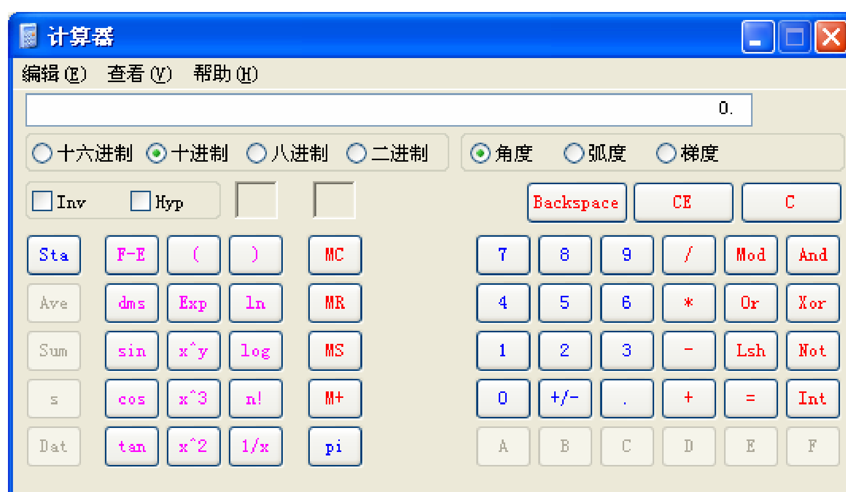


图 3-14 计算器

上图是普通计算器，还有一种科学性计算器，点击 查看->科学型，如图 3-15，得到我们的科学型计算器 3-16：



图 3-15 计算器



3-16 科学型计算器

这就包含进制计算和转换了，输入数据比较一下，图 3-17 先输入十进制 255，然后分别点击图 3-18　十六进制和图 3-19　二进制：



图 3-17 十进制



图 3-18 十六进制

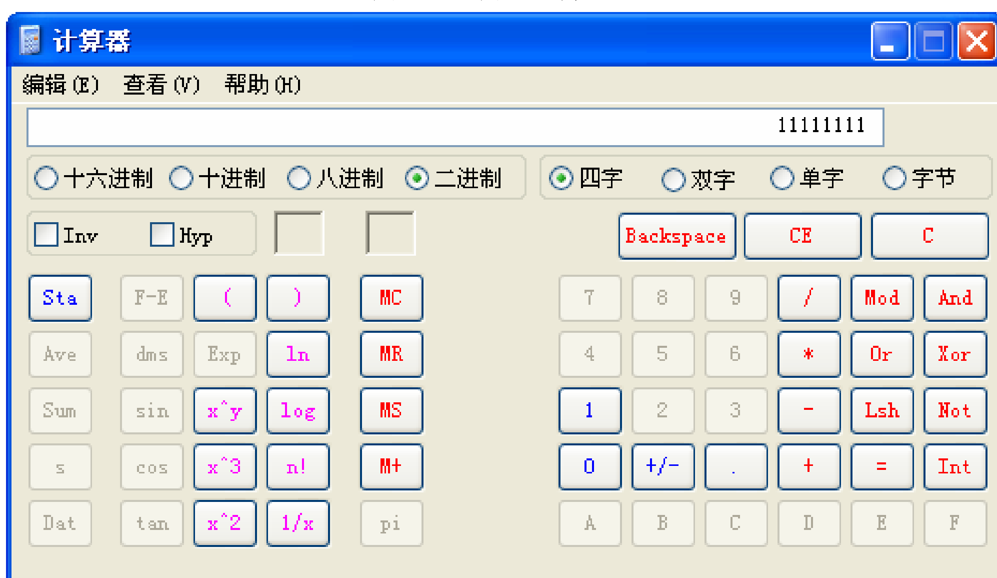


图 3-19 二进制

接下来，我们算一下 0xfe 这个数据如下图 3-20、图 3-21：



图 3-20 数据 0xfe 的十六进制表示方法

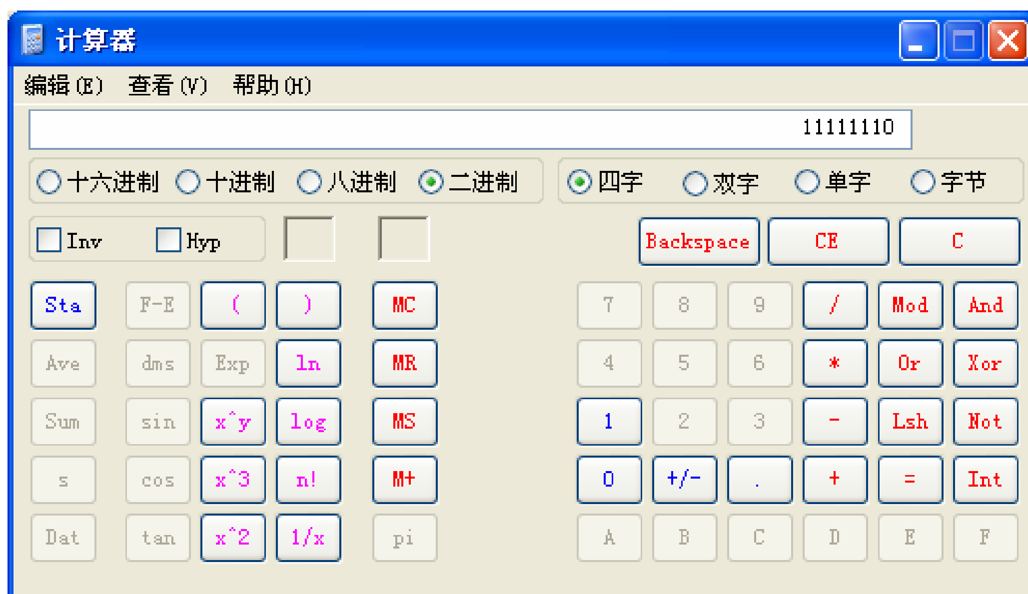


图 3-21 点击二进制数据 0xfe 的二进制表示方法

3.2.7 更多LED例程

更多 LED 相应的例程可通过北京航空航天大学出版社下载专区下载，如下表 3-5：

表 3-5 更多 LED 例程（含详细注释和文档分析）

序号	例程功能
例程 01	单片机 IO 输出-点亮 1 个 LED 灯方法 1
例程 02	单片机 IO 输出-点亮 1 个 LED 灯方法 2
例程 03	单片机 IO 输出-点亮 1 个 LED 灯方法 3
例程 04	单片机 IO 口输出-点亮多个 LED 灯方法 1
例程 05	单片机 IO 口输出-点亮多个 LED 灯方法 2

例程 06	单片机 IO 口输出-点亮多个 LED 灯方法 3
例程 07	单片机 IO 口输出-1 个 LED 闪烁
例程 08	单片机 IO 口输出-不同频率闪烁 1 个 LED 灯
例程 09	单片机 IO 口输出-不同频率闪烁多个 LED 灯
例程 10	8 位 LED 左移
例程 11	8 位 LED 右移
例程 12	LED 循环左移
例程 13	LED 循环右移
例程 14	查表显示 LED 灯
例程 15	双灯左移右移闪烁
例程 16	神奇美式花样灯
例程 17	PWM 逐渐变亮
例程 18	PWM 逐渐变暗
例程 19	PWM 调光
例程 20	LED 二进制加法
例程 21	LED 水滴程序

3.3 按键

3.3.1 按键的介绍

按键就是按下时闭合，松手后自动断开的器件，如下图3-22是几种单片机系统常见的按键。

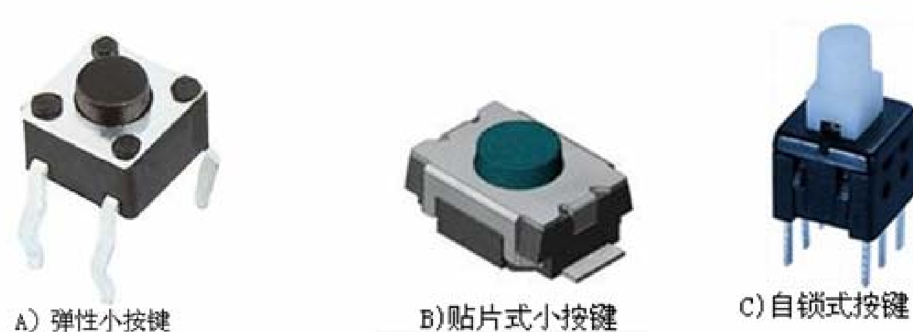


图 3-22 常见按键

按键又分为很多种，例如有一种是自锁式的按键，自锁式按键按下时闭合且会自动锁住，只有再次按下时才弹起断开。通常我们把自锁试按键当做开关使用，比如“神舟51+ARM”实验板上的电源开关就使用自锁按键。

单片机的外围输入控制用小弹性按键比较好，在神舟51单片机开发板中，我们把按键的一端接地，另一端与单片机的某个I/O口相连，开始时51单片机的I/O口始终为高电平，当按键闭合时按键将I/O口与地相连变成了低电平，此时程序一旦检测到I/O口变为低电平则说明按键被按下，就可以执行相应的0指令；在这里连接按键的单片机的I/O口被设置

成了输入模式，当I/O检测到低电平就会被触发，即按键已按下。

3.3.2 按键的抖动

什么是机械抖动？通常的按键开关为机械弹性开关，由于机械触点的弹性作用，一个按键开关在闭合时不会马上稳定地接通，在断开时也不会一下子断开。因而在闭合及断开的瞬间均伴随有一连串的抖动。抖动时间的长短由按键的机械特性决定，一般为5ms～10ms。这是一个很重要的时间参数，在很多场合都要用到；实际上只进行一次按键操作，但有可能执行了多次按键结果，这就是抖动造成的，所以大多数产品实际使用中都使用了按键去抖功能。

按键的连接方法和按键在被按下时其触点电压变化过程下图3-23：

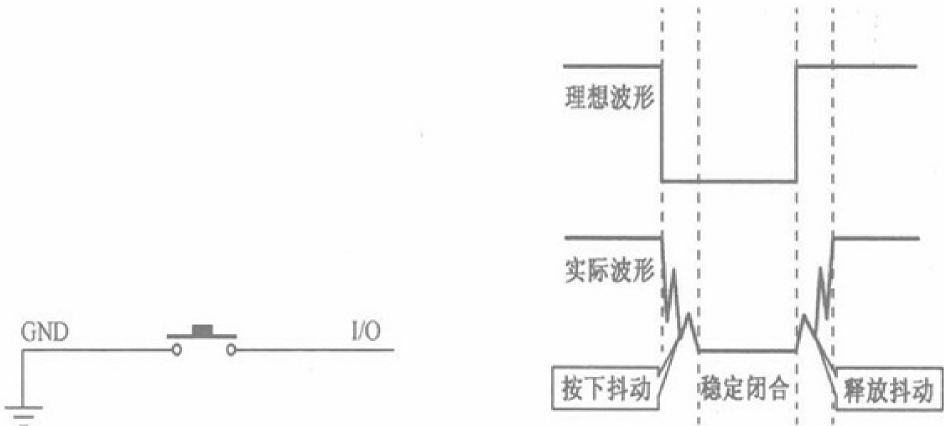


图 3-23 按键的连接方法与触点电压变化过程

这个抖动时间虽然很短，不同按键抖动不同，但对应单片机来说，很轻松就能检测到，单片机的运行的速度是微秒 us 级别。

用示波器跟踪一个小的按钮开关在闭合时的抖动现象，得到如下图3-24的波形。

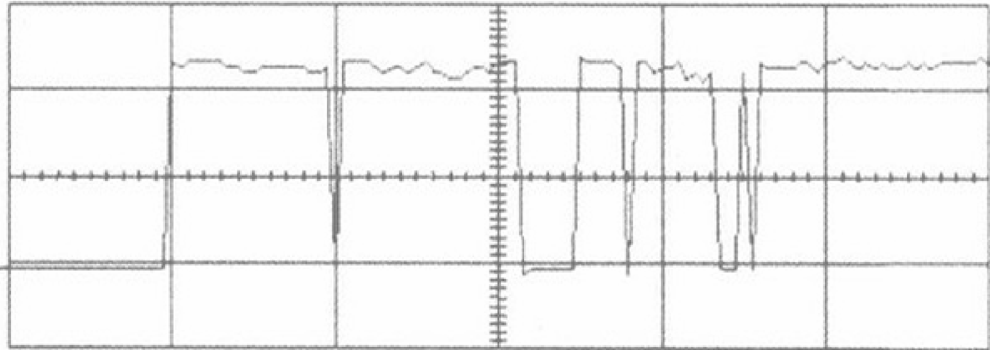


图 3-24 按键闭合抖动现象

观察波形可以帮助我们对抖动现象有一个直观的了解。水平轴2ms/Div，抖动间隙大约为10ms，在达到稳定状态前一共有6次变化，频率随时间升高。

那该如何去抖呢？通常情况下有两种去抖方法，一种是硬件去抖，另一种是软件去抖。

硬件去抖最简单的就是按键两端并联电容，使得一些抖动的毛刺被电容过滤掉，这里如果有不懂的读者可以网络搜索一下电容如何去除干扰的文章，这里的电容容量根据实验而定。当然电容去抖动是最常用也是最经济实惠的一种普通方案，要求更高的场景下也有专用的去抖动芯片。

软件去抖是通过延时来规避掉这些干扰，比如在某个阶段只取样一次，这样就可以避开那些干扰，例如人在 100ms 内只可能按一次按键，那么 100ms 内哪怕有 10 万次抖动，我们也只计算一次按下，这样使用不用增加任何硬件成本，容易调试，所以现在大都使用软件去抖。关于软件去抖的详细步骤如下，这是一种设计参考：

1）检测到按键按下后进行 5~10ms 延时，用于跳过这个抖动区域

2）延时后再检测按键状态，如果没有按下表明是抖动或者干扰造成，如果仍旧按下，可以认为是真正的按下。并进行对应的操作。

3）同样按键释放后也要进行去抖延时，延时后检测按键是否真正松开。

3.3.3 硬件原理图连接

按键硬件连接原理图如图 3-25 所示：

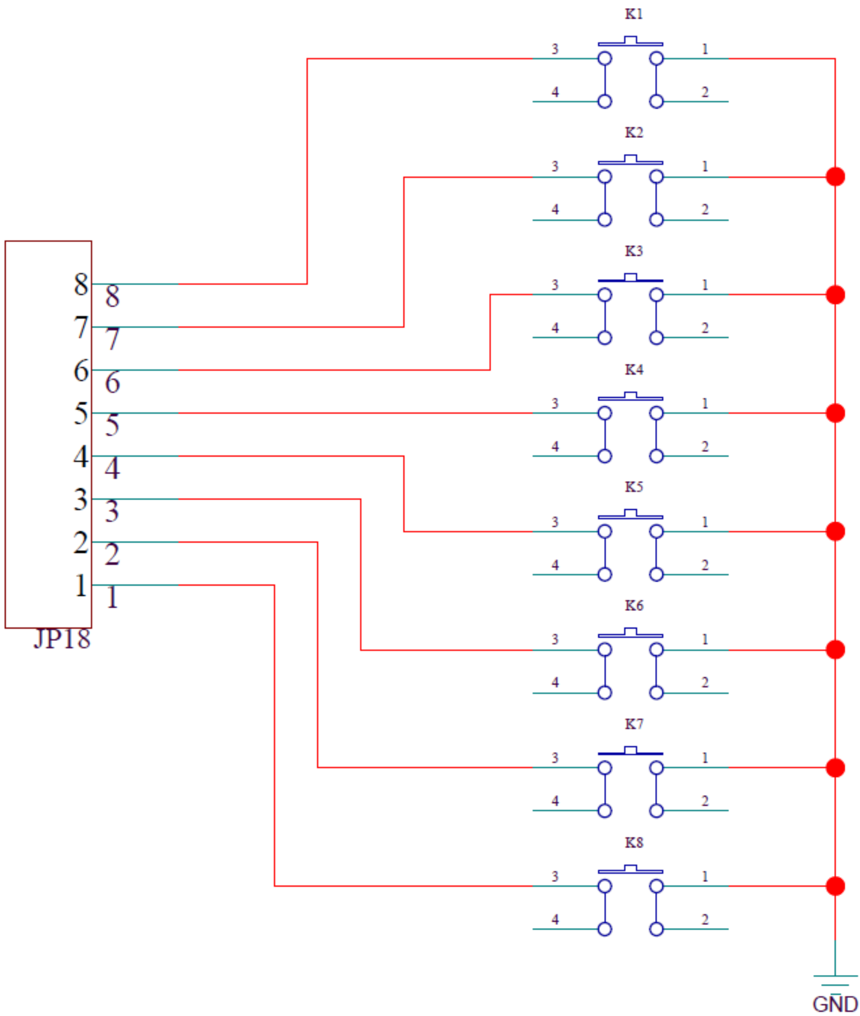


图 3-25 按键部分原理图

看原理图可以得知，单片机每个端口对应着一个按键，端口被赋值为高电平，即“1”，把对应的端口赋值低电平，即“0”，按键按下，通过检测该端口的电平即可判断按键是否按下。实验板上的按键实物图如下图 3-26 所示：

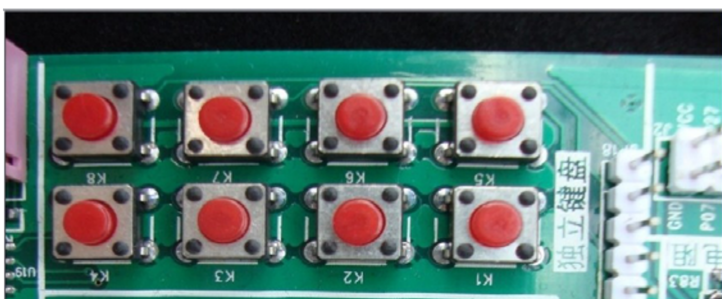


图 3-26　按键实物图

3.3.4 例程 01 一个独立按键控制LED（无消抖）

代码如下：

```
/*******************************************************
* 例程：单个独立按键检测
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：用于时刻检测按键状态并输出指示
*******************************************************/
#include<reg52.h> //包含头文件，一般情况不需要改动，头文件包含特殊功能寄存器的
定义
sbit KEY=P2^0;   //定义按键输入端口
sbit LED=P2^1;   //定义 led 输出端口
/*-- 主函数 --*/
void main (void)
{
    KEY=1;             //按键输入端口电平置高
    while (1)          //主循环
    {
        if(!KEY)       //如果检测到低电平，说明按键按下
            LED=0;
        else
            LED=1;  //这里使用 if 判断，如果按键按下 led 点亮，否则熄灭
    }
}
```

硬件连接如表 3-6

表 3-6 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P2.0 管脚	JP16	直连	JP18.0	1 根 1 针杜邦线	控制独立按键
P2.1 管脚	JP16	直连	JP19.0	1 根 1 针杜邦线	控制 LED 灯
实验现象：下载程序后，连好杜邦线，按下按键，灯就亮；松开按键，灯灭。					

知识要点：

从按键部分的原理图可以分析到，当按下按键的时候，使得管脚拉低，那么当 KEY=0 的时候，证明按键按下了，那么就点亮 LED 灯；当按键松开，KEY=1，灯不亮。

程序中的 LED 和 KEY 变量分别控制单片机的连按键的 IO 管脚和连灯的 IO 管脚，关于按键的原理图上面有，在原理图中，按键一端连的是 GND 地，一端连的是单片机的 IO 管脚，按键按下的时候，单片机的 IO 管脚将被拉低。关于 LED 灯的原理图请大家参见前面的程序。

3.3.5 例程 02 一个独立按键控制LED（消抖动）

代码如下：

```
/*****************************************************
* 例程：单个独立按键检测
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：按一次按键，led 点亮，再按一次熄灭，以此循环
******************************************************/
#include<reg52.h> //包含头文件，一般情况不需要改动，头文件包含特殊功能寄存器的
定义
sbit KEY=P2^0;   //定义按键输入端口
sbit LED=P2^1;   //定义 led 输出端口
void DelayUs2x(unsigned char t);//函数声明
void DelayMs(unsigned char t);
/*-- 主函数 --*/
void main (void)
{
    KEY = 1;             //按键输入端口电平置高
    while (1)            //主循环
    {
        if(!KEY)      //如果检测到低电平，说明按键按下
        {
            DelayMs(10);   //延时去抖，一般 10-20ms
            if(!KEY)       //再次确认按键是否按下，没有按下则退出
            {
                while(!KEY); //如果确认按下按键等待按键释放，没有释放一直等待
                {
                    LED=!LED; //释放则执行需要的程序
                }
            }
        }
    }
}
/*------------------------------------------------
 uS 延时函数，含有输入参数 unsigned char t，无返回值
 unsigned char 是定义无符号字符变量，其值的范围是
 0~255 这里使用晶振 12M，精确延时请使用汇编,大致延时
```

```
  长度如下 T=tx2+5 uS
------------------------------------------------*/
void DelayUs2x(unsigned char t)
{
    while(--t);
}
/*------------------------------------------------
  mS 延时函数，含有输入参数 unsigned char t，无返回值
  unsigned char 是定义无符号字符变量，其值的范围是
  0~255 这里使用晶振 12M，精确延时请使用汇编
------------------------------------------------*/
void DelayMs(unsigned char t)
{
    while(t--)
    {
        //大致延时 1mS
        DelayUs2x(245);
        DelayUs2x(245);
    }
}
```

硬件连接如表 3-7

表 3-7 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P2.0 管脚	JP16	直连	JP18.0	1 根 1 针杜邦线	控制独立按键
P2.1 管脚	JP16	直连	JP19.0	1 根 1 针杜邦线	控制 LED 灯
实验现象：下载程序后，连好杜邦线，按下按键，灯就亮；在按一下按键，灯变灭，反复循环					

知识要点：

1、按键去抖

首先用if(!KEY) 检测按键是否按下，如果没有则退出，如按下调用延时DelayMs(10)，延时过后重新检测按键状态，如果没有按下，说明是抖动或其他干扰误触发，放弃该结果，退出，反之，如果仍然按下，表明按键确实被按下，此时可以处理按键对应的程序。

所以按键的去抖操作就是延时了10-20ms时间。

2、按键等待释放

用语句while(!KEY)等待按键是否释放，这行程序可以确保每次按键只能作用一次，而且必须是在按键释放后再继续工作。

3、精准的延时

晶振是12M左右,那么指令周期1us,一个空操作指令nop延时1us,每执行一次while(--t)相当于执行了两条指令，那么花费时间2us。1而1ms = 1000us，那么我们在程序中需要达到10ms，就按这个计算就可以，具体请见程序代码。

3.3.6 更多按键的例程

更多按键相应的例程可通过北京航空航天大学出版社下载专区下载，如下表 3-8：

表 3-8 按键更多实验功能

序号	例程功能
例程 01	一个独立按键控制 LED（无抖动）
例程 02	一个独立按键控制 LED（消抖动）
例程 03	按键加减操作数码管显示
例程 04	多位数按键加减数码管显示（不闪烁）
例程 05	多位数按键加减数码管显示（闪烁）
例程 06	多位数按键加减数码管显示（完美版）
例程 07	定时器扫描数码管
例程 08	按键长按短按效果
例程 09	抢答器
例程 10	独立按键一次输入数据
例程 11	按键输入从左至右显示
例程 12	8 位端口检测 8 位独立按键

3.4 共阳数码管

3.4.1 共阳数码管的介绍

按发光二极管单元连接方式可分为共阳极数码管和共阴极数码管。共阳数码管是指将所有发光二极管的阳极接到一起形成公共阳极(COM)的数码管，共阳数码管在应用时应将公共极 COM 接到+5V，当某一字段发光二极管的阴极为低电平时，相应字段就点亮，当某一字段的阴极为高电平时，相应字段就不亮。共阴数码管是指将所有发光二极管的阴极接到一起形成公共阴极(COM)的数码管，共阴数码管在应用时应将公共极 COM 接到地线 GND 上，当某一字段发光二极管的阳极为高电平时，相应字段就点亮，当某一字段的阳极为低电平时，相应字段就不亮。

数码管按段数可分为七段数码管和八段数码管，八段数码管比七段数码管多一个发光二极管单元（多一个小数点显示）；按能显示多少个“8”可分为 1 位、2 位、3 位、4 位、5 位、6 位、7 位等数码管。

3.4.2 共阳数码管的内部原理

LED 数码管（LED Segment Displays）是由多个发光二极管封装在一起组成“8”字型的器件，引线已在内部连接完成，只需引出它们的各个笔划，公共电极。如下图 3-27 所示：

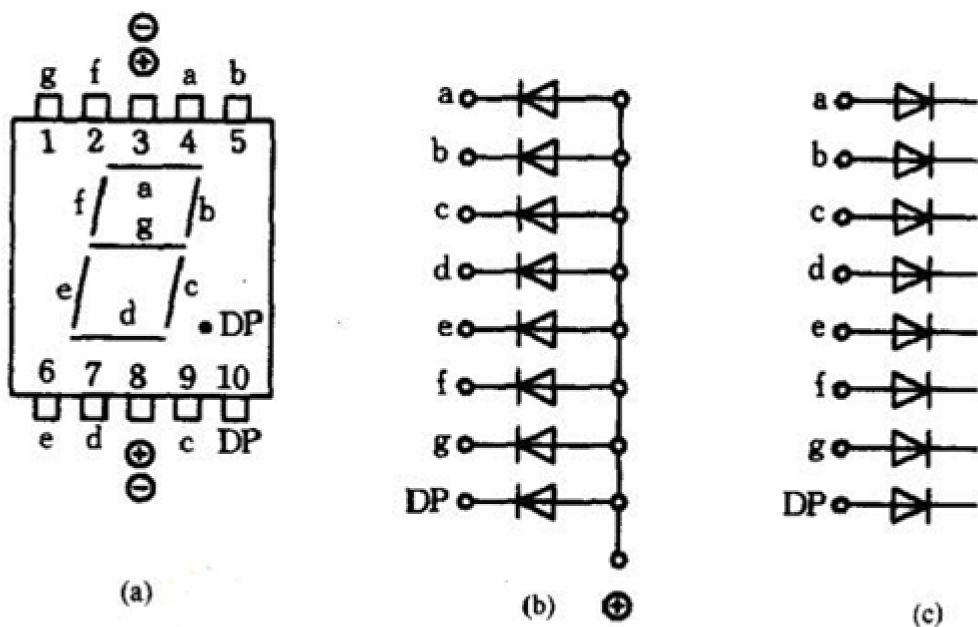


图 3-27 数码管信号分布与内部连接图

图 b 为数码管的内部接线图，图中把 8 个 LED 的阳级（正极）接在一起，这种接法的数码管称为共阳级数码管；另外一端 abcdefg、dp 信号接低电平，二极管就会点亮，数码管对应的那个位置的灯就会亮起。

3.4.3 共阳数码管的硬件连接原理

数码管要正常显示，就要用驱动电路来驱动数码管的各个段码，从而显示出我们要的数字，共阳极就是公共端连到 VCC 上，共阴极就是公共端连到 GND 上，具体可以参考下图 3-28 共阳与共阴的硬件连接图：

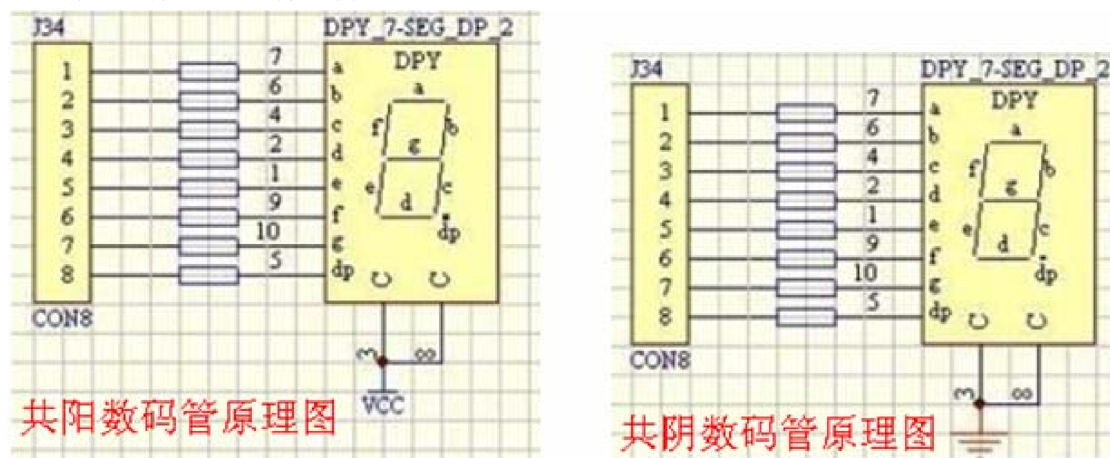


图 3-28 共阳与共阴的硬件连接图

下图 3-29 为数码管的内部连线图：

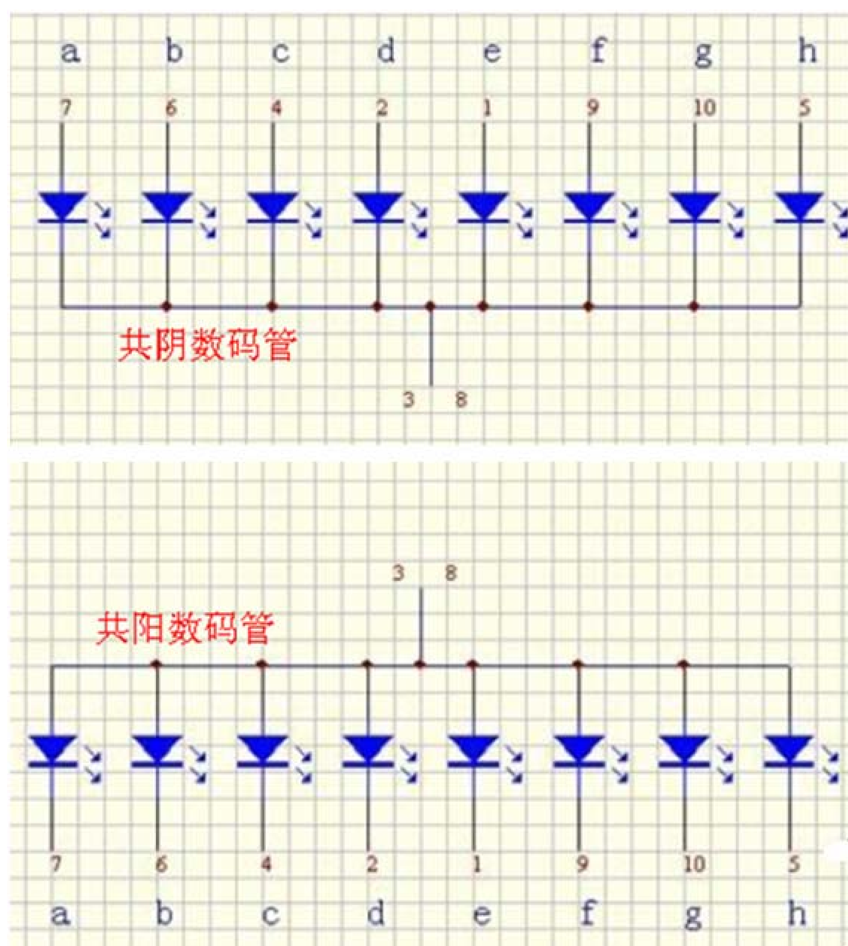


图 3-29 数码管内部连线图

单片机的 I/O 口连接到 JP24 的座子后经过电阻连接到数码管，因为该数码管为共阳数码管，所以数码管的公共端 COM 需要接到 VCC，只要单片机输出低电平就能点亮数码管。每个数码管的每一个段码都由一个单片机的 I/O 端口进行驱动，或者使用如 BCD 码二-十进制译码器译码进行驱动。优点是编程简单，显示亮度高，缺点是占用 I/O 端口多；下图 3-28 所示为神舟 51+ARM 数码管部分共阳的硬件连接图，JP34 为单片机 I/O 口的连接座子。

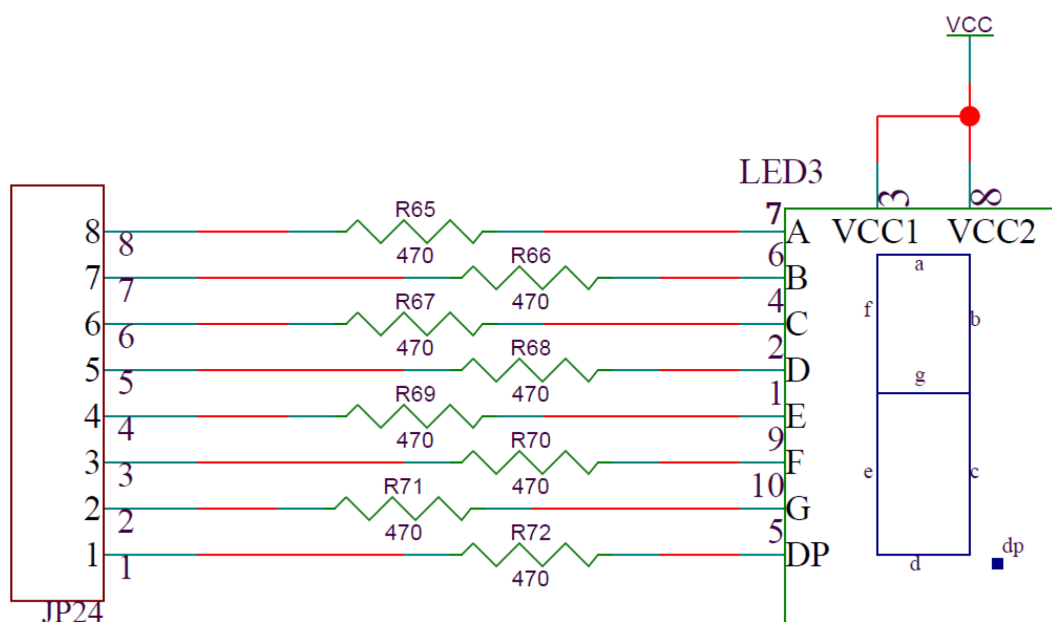


图 3-30 数码管部分原理图

3.4.4 例程 01 共阳数码管静态显示数字 8

代码如下：

```
/*****************************************************
* 例程：IO 口高低电平控制点亮一个 LED 灯
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：通过赋值给 P2，让数码管显示一个数字 8
*****************************************************/
#include<reg52.h> //包含头文件，一般情况不需要改动，
//头文件包含特殊功能寄存器的定义

void main (void)
{
// 参考数码管原理图，单片机 P2 的 IO 管脚为 0 时对应数码管为亮，为 1 的时候熄灭
    P2=0x0; //二进制 为 0000 0000
}
```

硬件连接如表 3-9

表 3-9 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P2 口	JP16	直连	JP24	1 根 8 针扁平电缆	控制共阳数码管
实验现象：下载程序后，连好杜邦线，数码管显示“8”					

3.4.5 例程 02 共阳数码管静态显示数字 0

代码如下：

```
/*****************************************************
* 例程：IO 口高低电平控制点亮一个 LED 灯
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：通过赋值给 P2，让数码管显示一个数字 0
*****************************************************/
#include<reg52.h>     //包含头文件，一般情况不需要改动，
                      //头文件包含特殊功能寄存器的定义
void main (void)
{
 // 参考数码管原理图，单片机 P2 的 IO 管脚为 0 时对应数码管为亮，为 1 的时候熄灭
```

```
    P2=0xc0;  //二进制 为 1100 0000
}
```

硬件连接如表 3-10

表 3-10 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P2 口	JP16	直连	JP24	1 根 8 针扁平电缆	控制共阳数码管
实验现象：下载程序后，连好杜邦线，数码管显示“0”					

知识要点：

我们可以看下图 3-31，实际要显示一个 0，就是使得 g 和 dp 不显示，其他 abcdef 都显示就可以了，所以我们使得两位为高电平，其他六位为低电平即可，1100 0000 这里的两个 1 分别代表了 g 和 dp，所以最后再我们的共阳数码管上显示出了字符“0”。

显示字型	dp ,g,f,e,d,c,b, a	字符码
0	1 1 0 0 0 0 0 0	C0H
1	1 1 1 1 1 0 0 1	F9H
2	1 0 1 0 0 1 0 0	A4H
3	1 0 1 1 0 0 0 0	B0H
4	1 0 0 1 1 0 0 1	99H
5	1 0 0 1 0 0 1 0	92H
6	1 0 0 0 0 0 1 0	82H
7	1 1 1 1 1 0 0 0	F8H
8	1 0 0 0 0 0 0 0	80H

图 3-31 0~9 字符码

3.4.6 例程 03 共阳数码管循环显示数字 0-9

代码如下：

```
/********************************************************
* 例程：IO 口高低电平控制点亮一个 LED 灯
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：通过循环赋值给 P2，让数码管显示特定的字符或者数字
********************************************************/
#include<reg52.h> //包含头文件，一般情况不需要改动，头文件包含特殊功能寄存器的
定义
unsigned char code table[10]={0xc0,0xf9,0xa4,0xb0,0x99,0x92,0x82,0xf8,0x80,0x90,};
// 显示数值表 0-9
void Delay(unsigned int t); //函数声明
```

```
/*------------------------------------------------
                主函数
------------------------------------------------*/
void main (void)
{
    unsigned char i;   //定义一个无符号字符型局部变量 i 取值范围 0~255

    while (1)          //主循环
    {
        for(i=0;i<10;i++) //加入 for 循环，表明 for 循环大括号中的程序循环执行 10 次
        {
            P2=table[i];        //循环调用表中的数值
            Delay(50000);       //延时，方便观看数字变化
        }
    }
}
/*------------------------------------------------
 延时函数，含有输入参数 unsigned int t，无返回值
 unsigned int 是定义无符号整形变量，其值的范围是
 0~65535
------------------------------------------------*/
void Delay(unsigned int t)
{
 while(--t);
}
```

硬件连接如表 3-11

表 3-11 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P2 口	JP16	直连	JP24	1 根 8 针扁平电缆	控制共阳数码管
实验现象：下载程序后，连好杜邦线，数码管循环显示数字 0，1，2 一直到 9					

知识要点：

1. 我们将数字 0-9 存储到了 table[]的数组里，可参考一下图 3-31 这个 0-9 具体对应如下这张图表 3-31 中，table[i]的一个成员就是 16 进制数，例如 0xc0，变成二级制后就是 1100 0000，直接将该值赋值给 P2 端口。

2. unsigned char code table[]表示这个数组是静态的

指定数组 table[]存储在 code 区，也就是 rom 或者是 flash，这要看单片机程序存储器的构成；code 去掉也可以，不过是把变量存放在别的区域。这要根据编译器的设定存储模式来定。

如果变量体积大了，就要定义在 code 区。你放在 RAM 区，也是需要在 ROM 里占用同样的大小，不然，上电时候，你 RAM 里指定数据从何而来？反之，放在 ROM 的数据，只有在调用的时候才占用 RAM 容量，毕竟 51 系列的 RAM 是小的，尤其是程序比较大的时候，不够用。

3.4.7 更多共阳数码管例程

更多共阳数码管相应的例程可通过北京航空航天大学出版社下载专区下载，如下表3-12：

表 3-12 共阳数码管更多实验功能（含详细注释和文档分析）

序号	例程功能
例程 01	共阳数码管静态显示数字 8
例程 02	共阳数码管静态显示数字 0
例程 03	共阳数码管循环显示数字 0-9
例程 04	共阳数码管模拟流水灯
例程 05	按键控制单个共阳数码管 1
例程 06	按键控制单个共阳数码管 2
例程 07	单个共阳数码管指示高 H 或低 L

3.5 共阴数码管

3.5.1 八位共阴数码管简介

数码管由七个条状和一个点状发光二极管管芯制成，称为七段数码管。根据其结构的不同，可分为共阳极数码管和共阴极数码管两种如图 3-32 所示。 共阳共阴，是针对数码管的公共脚而说的。典型的一位数码管，一般有 10 个脚，8 个段码（7 段加 1 个小数点），剩下两个脚接在一起。各个段码实际上是一个发光二极管，既然是发光二极管，就有正负极。共阴是公共脚是负极（阴极），段码位是阳极，当公共脚接地，段码位接高电平时，对应段码位点亮。一位数码管就是这样，多位的数码管原理类似。

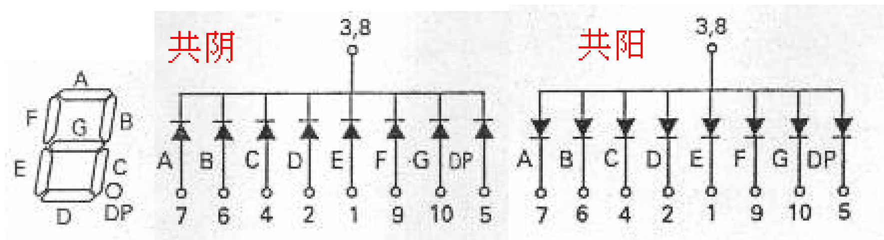


图 3-32 单个数码管结构图

多位数码管也是一样，内部把各位的数码管 8 个显示笔划"a,b,c,d,e,f,g,dp"的同名端连在一起，每个数码管的公共极 COM 单独引出来。我们开发板上八位共阴数码管为两个四联共阴数码管组成。四联共阴数码管一般共 12 个脚，4 个数码管共阴极分开，4 个数码管的段码并联，其结构图如下图 3-33：

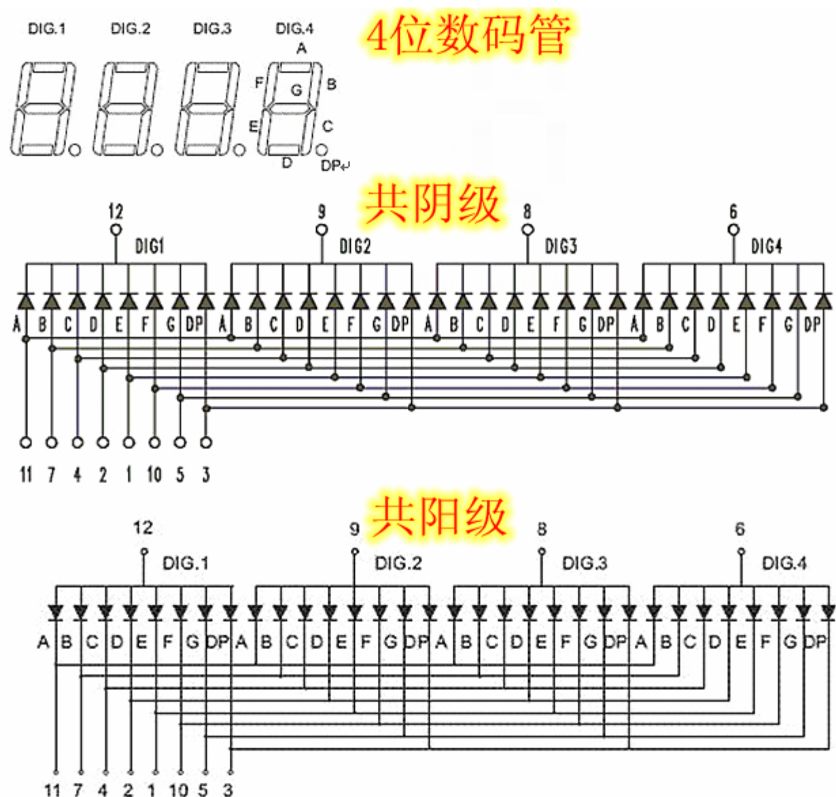


图 3-33 四联数码管段码图

3.5.2 八位共阴数码管的工作方式

8 位数码管要正常显示，就要用驱动电路来驱动数码管的各个段码，从而显示出我们要的数字，因此根据数码管的驱动方式的不同，分为静态式和动态式两类。下面我们来回顾一下这 2 个驱动方式

◆ 静态显示驱动

静态驱动也称直流驱动。静态驱动是指每个数码管的每一个段码都由一个单片机的 I/O 端口进行驱动，或者使用如 BCD 码二-十进制译码器译码进行驱动。静态驱动的优点是编程简单，显示亮度高，缺点是占用 I/O 端口多，如驱动 5 个数码管静态显示则需要 5×8=40 根 I/O 端口来驱动，要知道一个 89S51 单片机可用的 I/O 端口才 32 个呢：)，实际应用时必须增加译码驱动器进行驱动，增加了硬件电路的复杂性。

◆ 动态显示驱动

数码管动态显示接口是单片机中应用最为广泛的一种显示方式之一，动态驱动是将所有数码管的 8 个显示笔划"a,b,c,d,e,f,g,dp"的同名端连在一起，另外为每个数码管的公共极 COM 增加位选通控制电路，位选通由各自独立的 I/O 线控制，当单片机输出字形码时，所有数码管都接收到相同的字形码，但究竟是哪个数码管会显示出字形，取决于单片机对位选通 COM 端电路的控制，所以我们只要将需要显示的数码管的选通控制打开，该位就显示出字形，没有选通的数码管就不会亮。通过分时轮流控制各个数码管的的 COM 端，就使各个数码管轮流受控显示，这就是动态驱动。在轮流显示过程中，每位数码管的点亮时间为 1～2ms，由于人的视觉暂留现象及发光二极管的余辉效应，尽管实际上各位数码管并非同时点亮，但只要扫描的速度足够快，给人的印象就是一组稳定的显示数据，不会有闪烁感，动态

显示的效果和静态显示是一样的，能够节省大量的 I/O 端口，而且功耗更低。

3.5.3 硬件原理图连接

硬件连接如下图 3-34

图 3-34 8 位数码管硬件连接原理图

将 J17 短接 VCC 端，将锁存器配置为直通模式，用一根 8 位排线将板上的 JP23 插针的与单片机的 P0 口 JP15 插针连接，用另一根 8 位排线将板上的 JP22 插针的与单片机的 P2 口 JP16 插针连接，如下图 3-35：

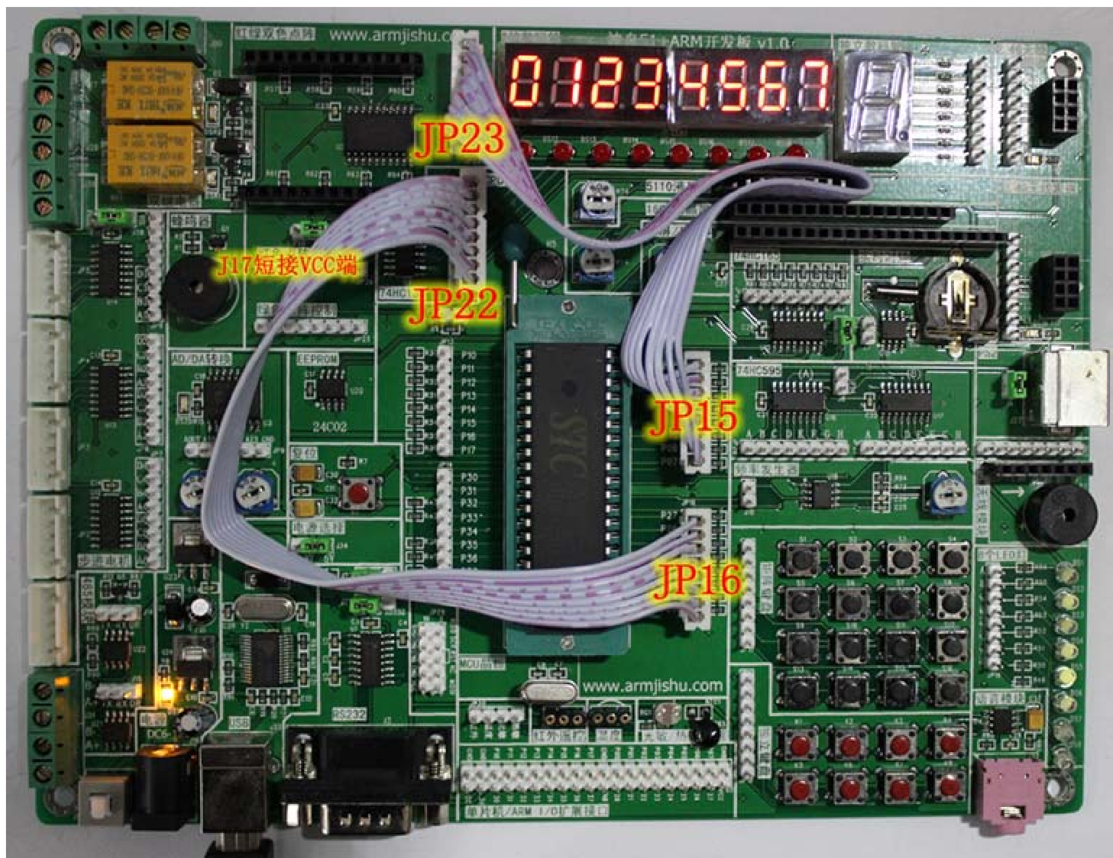


图 3-35 硬件连接实物图

硬件连接如表 3-13

表 3-13 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P0	JP15（A 向左）	直连	JP23(A 向右）	8 位排线	8 位数码管段码控制
P2	JP16（A 向右）	直连	JP22(A 向右）	8 位排线	8 位数码管段位控制
注意：将 J17 短接 VCC 端，将锁存器配置为直通模式					

3.5.5 例程 01 八位数码管显示其中之一

代码如下：

```
/*******************************************************
* 例程：8 位数码管显示其中之一
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：8 位数码管中其中一位显示
*******************************************************/
#include<reg52.h> //包含头文件，一般情况不需要改动，
                  //头文件包含特殊功能寄存器的定义
/*------------------------------------------------
                  主函数
```

```
------------------------------------------------*/
void main (void)
{
        while(1)
        {
                P2 = 0xFE;      //取位码，第一位数码选通，即二进制 1111 1110
                P0 = 0x4F;      //取显示数码，段码“3”共阴字符码
        }
}
```

硬件连接如表 3-14

表 3-14 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P0	JP15（A 向左）	直连	JP23(A 向右)	8 位排线	8 位数码管段码控制
P2	JP16（A 向右）	直连	JP22(A 向右)	8 位排线	8 位数码管段位控制
注意：将 J17 短接 VCC 端，将锁存器配置为直通模式					
实验现象：下载程序后，我们可以看到第 1 个数码管显示“3”					

知识要点：

我们通过单片机的 P0 口控制 8 位数码管段码值为“3”，再通过单片机的 P1 口控制 8 位数码管段码的位，选择第一个数码管，这样最后在第 1 个数码管显示“3”。

3.5.6 更多有关共阴数码管例程

更多共阴数码管相应的例程可通过北京航空航天大学出版社下载专区下载，如下表 3-15：

表 3-15 更多八位数码管丰富例程（含详细注释和文档分析）

序号	例程功能
例程 01	八位数码管显示其中之一
例程 02	八位数码管显示其中之二
例程 03	八位数码管动态扫描显示
例程 04	八位数码管动态扫描原理显示
例程 05	数码管显示动态数据
例程 06	1 个共阴数码管累加
例程 07	2 个共阴数码管累加
例程 08	3 个共阴数码管累加
例程 09	4 个共阴数码管累加
例程 10	1 个共阴数码管累减
例程 11	2 个共阴数码管累减
例程 12	3 个共阴数码管累减
例程 13	4 个共阴数码管累减
例程 14	共阴数码管显示小数点
例程 15	共阴数码管消隐

例程 16	共阴数码管递加递减带消隐
例程 17	共阴数码管左移
例程 18	共阴数码管右移 1
例程 19	共阴数码管右移 2

3.5 定时器

3.5.1 名词解释

1．时钟周期

时钟周期也称为震荡周期, 定义为时钟脉冲的倒数, 是计算机中最基本, 最小的时间单位; 在一个时钟周期内, CPU 只完成最基本的动作. 对同一种机型而言, 时钟频率越高, 计算机工作速度越快。例如，开发板上晶振是 11.0592MHZ 的，那么时钟周期=1/11059200 秒。

2．机器周期

在计算机中, 为了便于管理, 通常把一条指令执行划分为若干个阶段, 每一个阶段完成一项任务；如: 取指令, 存储器读, 存储器写等, 这每一项工作称为一个基本操作.，完成一个基本操作所需要的时间为机器周期, 一个机器周期由若干个 S 周期(状态周期)组成。51 单片机一个机器周期是 12 个时钟周期。

3．指令周期

执行一条指令所需要的时间, 一般由若干个机器周期组成. 指令不同,所需要的机器周期也不同。对于一些简单的单字节指令, 在取指令周期中, 指令取出到指令寄存器后, 立即译码执行, 不再需要其他的机器周期。

对一些比较复杂的指令, 例如:转移指令, 乘法指令, 则需要两个或两个以上的机器周期.。通常含一个机器周期的指令称为单周期指令, 包含两个机器周期的指令称为双周期指令。

4．总线周期

由于存储器和 I/O 是挂接在总线上的, CPU 对存储器和 I/O 的访问是通过总线进行的。通常把 CPU 通过总线对微处理器外部(存储器或 I/O 端口)进行一次访问所需要时间称为一个总线周期。

5．几个周期的不同之处

1）时钟周期是最小单位（CPU 晶振的工作频率的倒数）

2）机器周期需要 1 个或多个时钟周期（在 51 单片机中是 12 个时钟周期，硬件规定的）

3）指令周期需要 1 个或多个机器周期

4）机器周期因涉及一个基本操作时间, 可能操作总线, 因此可能会包含总线周期,也可能不包含。

例如时钟周期这个单位时间是由单片机上连接的晶振来提供的，比如神舟 51 单片机开发板上的晶振是 11.0592MHz，时钟周期为晶振频率的倒数，而机器周期是 12 个时钟周期组成。机器周期 = 12 x (1/11059200) = 1.0851us。

注意：定时器是按机器周期来计数的，所以例如要技术 5ms 时钟的数字，定时器该设

定一个怎样的初值呢？5ms/1.0851us = 5000/1.0851 = 4607 次；也就是说，在 51 单片机中使用 11.05926 的晶振，定时器设定 4607 这个数，每个机器周期都减 1，一直减到 0，就是经历了 5ms 的时钟。

3.5.2 定时器的由来

定时器为一个计算时间的设备。人类最早使用的定时工具是沙漏或水漏，后来逐渐发展成了我们现在的电子表、机械表等。它有倒计时与顺时计时 2 种定时方式，像我们的手表使用的就是顺时计时的，比赛用的定时方式大多是倒计时的。当定时器将时间累加到我们设定的数值时，它就会发送一个信号到控制端，如单片机，相当于一个开关装置。

随着单片机的产生，单片机设定了一种定时器/计数器的功能，根据定时时间的大小；另外，定时器还配置了几种工作方式，可以根据不同的情况选择合适的定时器工作方式。

3.5.2 定时器实现原理与作用

通俗的说：单片机的定时器工作原理如同一个盛水的盆子，根据不同的设定（工作模式 0,1,2,3）盆子的大小不同，而接水的方式却是相同的（时钟周期），为一滴一滴的接水，比如，在某种工作模式下，接满一盆水要 1000 滴，每滴水用时 1 秒钟，此时接满一盆水要用 1000 秒时间；于是，水滴数（计数值）与时间就有了相对的关系，但一定要记得，盆里的水永远是满的，如果我们要计时 50 秒，那么我们就要先在盆里倒出 50 滴水，而后开始接水，当盆里的水满了并且溢出时，单片机会提示，已经计时 50 秒了，请关闭水源或做其他处理。

单片机中的定时器和计数器其实是同一个物理的电子元件，只不过计数器记录的是单片机外部发生的事情(接受的是外部脉冲)，而定时器则是由单片机内部对数字做减法，定时器开始时，先设定一个数值，一个单位时间减少 1，一直减少到 0 为止；到 0 之后就会产生一个中断异常，通知 CPU 采取相应的处理，此时根据设定通常是重新再装载一次初值，定时器又从初始值开始逐渐递减到 0，如此循环，程序可以设定定时器初值的大小也就是发生一次定时器中断的时间，记录多少次循环就达到了一个整数的延时时间。

3.5.4 定时器的四种不同工作方式

51 系列单片机有两个定时器：T0 和 T1，这两个定时器都是 16 位的定时器/计数器，T0 由特殊功能寄存器 TH0 和 TL0 构成,而 T1 则是由 TH1 和 TL1 构成;定时器的结构图如下图 3-36 所示。

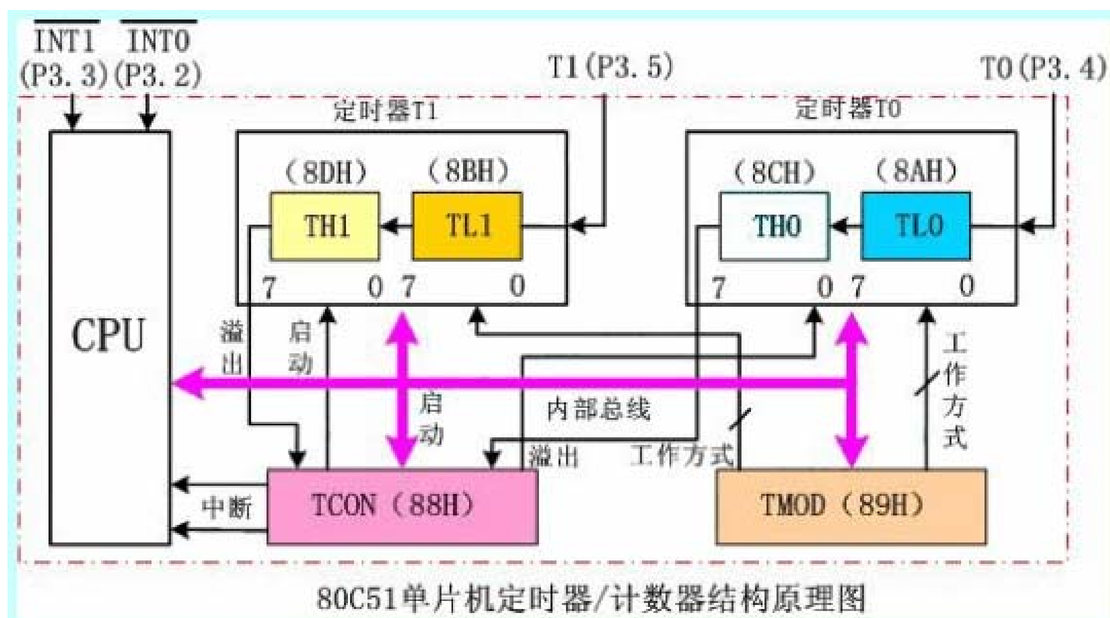


图 3-36 单片机定时器结构图

两个寄存器 TCON（控制寄存器）与 TMOD（工作方式寄存器）可由软件设置来配置 T0 和 T1 定时器；定时器工作不占用 CPU 时间，除非定时器溢出，才能中断 CPU 的当前操作。每个定时器还有四种工作模式，其中模式 0，模式 1，模式 2 对 T0 和 T1 都一样，模式 3 对两者不同。下面我们详细来了解一下定时器的 4 种工作方式：

1）工作模式 0:

由 TL0 的低 5 位和 TH0 的全部 8 位共同构成一个 13 位的定时器/计数器;定时器/计数器启动后,定时或计数脉冲个数加到 TL0 上,从预先设置的初值(时间常数)开始累加,不断递增 1;当 TL0 计满后,向 TH0 进位,直到 13 位寄存器计满溢出;溢出时,定时器/计数器硬件会自动地把 13 位的寄存器值清 0,中断标记 TF0 置 1;如果需要进一步定时/计数,需要使用相关指令重置时间常数,并把定时器/计数器的中断标记 TF0 置 0;工作模式 0 的结构如下图 3-37:

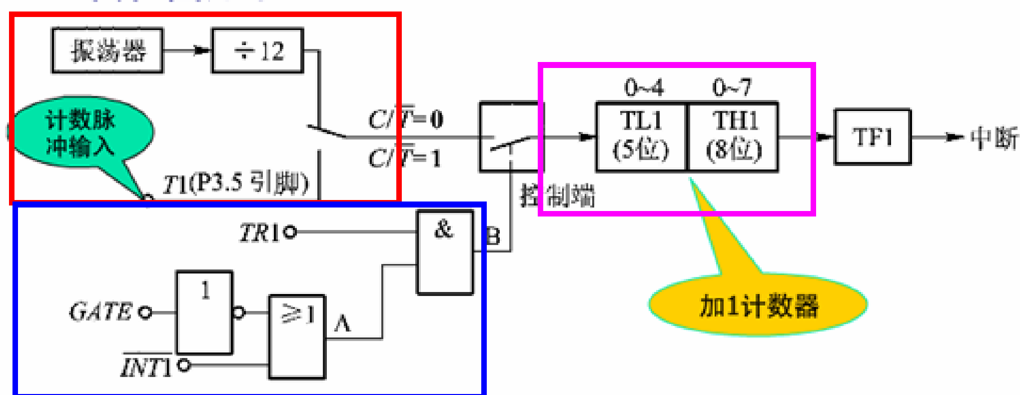


图 3-37 工作模式 0

图中，红色方框的功能是为得到一个振荡周期，C/T 位为 0 时为定时方式。蓝色方框是寄存器的配置方式，紫色方框是 16 位寄存器的使用情况，如下图 3-38 所示，当定时时间到时，计数溢出时，TF1 置 1，产生中断请求。

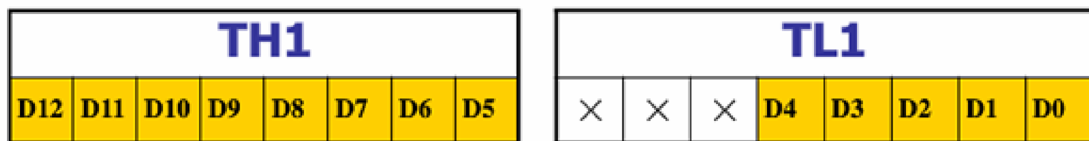


图 3-38 16 位寄存器使用情况

2）工作模式 1:

模式 1 与模式 0 几乎完全相同,唯一的区别就是，模式 1 中的寄存器 TH0 和 TL0 共同构成的是一个 16 位定时器/计数器来参与操作，因此比模式 0 中的定时/计数范围更大；工作模式 1 的结构如下图 3-39:

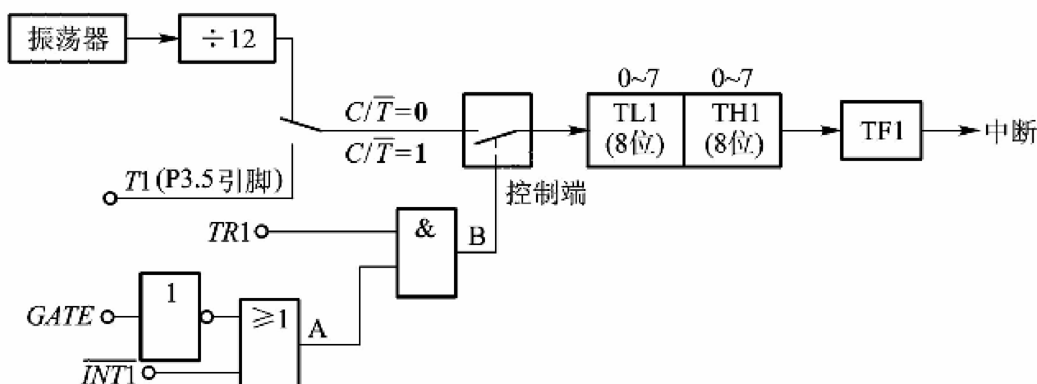


图 3-39 工作模式 1

3）工作模式 2:

这种模式又称为自动再装入预置数模式；当定时器/计数器的寄存器 TH0/TL0 的值溢出时,定时器/计数器硬件设备会自动把寄存器 TH0/TL0 的值清 0，以重新开始操作；但是有时候,我们的定时/计数操作是需要多次重复定时/计数的,如果溢出时不做任何处理，那么，在第二轮定时/计数时就是从 0 开始定时/计数了，而这并不是我们想要的；所以，要保证每次溢出之后,在重新开始定时/计数的操作是我们想要的,那就要把预置数(时间常数)重新装入某个地方;而重新装入预置数的操作是硬件设备自动完成的，不需要人工干预所以,这种工作模式就叫自动再装入预置数方式;既然需要重新装入预置数，那么预置数就必须要存放在某个地方,才能保证重装操作成功;在工作模式 2 中，把自动重装入的预置数存放在定时器/计数器的寄存器的高 8 位中，也就是存放在 TH0 中，而只留下 TL0 参与定时/计数操作；显然，定时/计数的方位小了很多。

这个工作模式常用于波特率发生器(串口通讯),T1 工作在串口模式 2;用于这种方式时,定时器就是为了提供一个时间基准;计数溢出之后,不需要做太多的事情,只做一件事就可以,就是重新装入预置数,再开始重新计数,而且中间不需要任何延时;工作模式 2 的结构如下图 3-40:

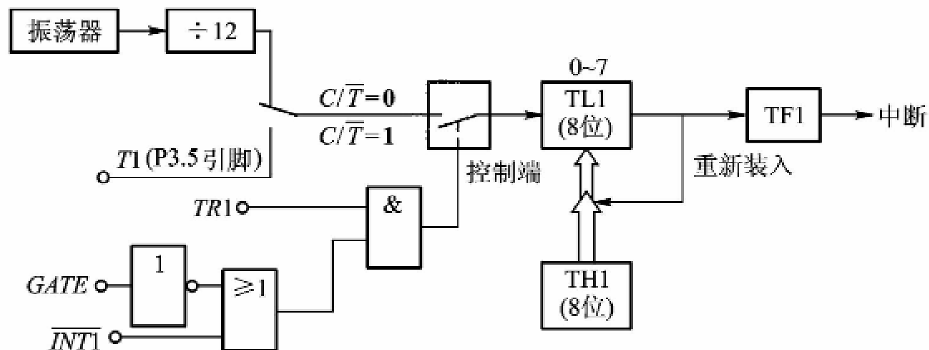


图 3-40 工作模式 2

4）工作模式 3:

由于定时器/计数器 T1 没有工作模式 3,如果把定时器/计数器 T0 设置为工作模式 3,那么 TL0 和 TH0 将被分割成两个相互独立的 8 位定时器/计数器;工作模式 3 的结构如下图 3-41:

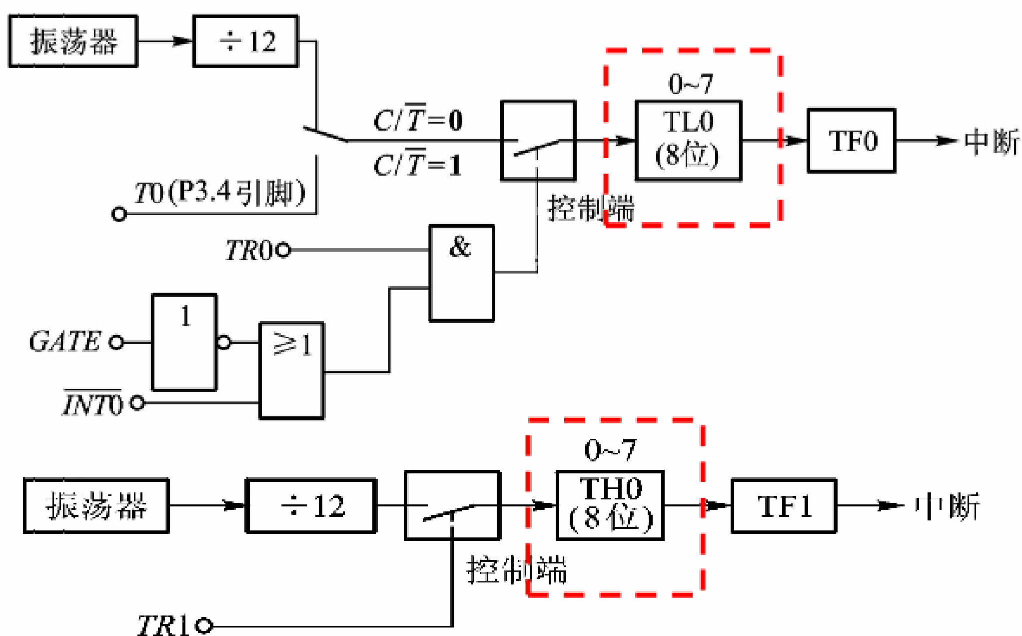


图 3-41 工作模式 3

5）定时器的定时范围:

工作方式 0——13 位定时器工作 模式,最多可计数 2 的 13 次方 次,即:8192 次,[0,8191];
工作方式 1——16 位定时器工作 模式,最多可计数 2 的 16 次方 次,即:65536 次,[0,65535];
工作方式 2——8 位定时器工作模式，计算次数最多为 2^8,即 256，,[0,255];
工作方式 3——8 位定时器/工作模式 ，计算次数最多为 2^8,即 256，,[0,255];

6）定时/计数初值计算方法:

定时/计数初值=最大值-需要需要定时/计数的时间次数;

3.5.6 例程 01 用定时器使得LED灯闪烁

例程实现定时器 0 与定时器 1 控制单片机板子上的 LED 灯的运行状态，到一定的时间后控制一次，实现定时的效果，还用定时器产生各频率周期的方波。通过这几个例程，让我们能对定时器有一个全新的认识，实现一个理论与实践相结合的效果，不再是纸上谈兵。

代码如下：

```
/*****************************************************
* 例程：IO 口高低电平控制点亮一个 LED 灯
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：左移，直至 LED 全部点亮，左移符号
*****************************************************/
#include<reg52.h>
                //包含头文件，一般情况不需要改动，头文件包含特殊功能寄存器的定义
sbit LED=P2^2;      //定义 LED 端口
/*- 定时器初始化子程序 -*/
void Open_Timer0(void)
{
    TMOD |= 0x01; //使用模式 1，16 位定时器，使用"|"符号可以在使用多个定时器时
不受影响
    TH0=0x00;   //给定初值，这里使用定时器最大值从 0 开始计数一直到 65535 溢出
    TL0=0x00;
    EA=1;           //总中断打开
    ET0=1;          //定时器中断打开
    TR0=1;          //定时器开关打开
}
/*- 定时器中断子程序 -*/
void Timer0(void) interrupt 1 using 1
{
    TH0=0x00;           //重新赋值
    TL0=0x00;
    LED=~LED;           //指示灯反相，可以看到闪烁
}
/*------------------------------------------------
                主程序
------------------------------------------------*/
main()
{
    Open_Timer0();
    while(1);
}
```

硬件连接如表 3-16：

表 3-16 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P2 口	JP16(A 向左)	直连	JP19（A 向右）	1 根 8 针扁平电缆	控制 LED 灯
实验现象：下载程序后，连好杜邦线，有 1 个 LED 灯不停的闪烁（定时器每次到时就会对 LED 灯取反，比如原本是亮的状态就会变灭的状态；原本是灭的状态就会变亮）					

硬件连接实物图如图 3-42：



图 3-42 硬件连接实物图

知识要点：

该例程代码是通过定时器 0 控制 LED 灯按一定频率亮灭操作的功能，下面我们看下程序代码是如何完成这个操作的。

1．函数 Open_Timer0()使用 TMOD 寄存器设置使用定时器 1，然后对定时器 1 赋初值，然后把单片机内部的总中断打开，定时器中断打开，定时器开关打开，这些都是 51 单片机芯片内部协议规定的，只需要按照规定打开这些就可以，值分析的是 TMOD 寄存器到底该如何设置。

```
void Open_Timer0(void)
{
    TMOD |= 0x01; //使用模式 1，16 位定时器，使用"|"符号可以在使用多个定时器时不受影响
    TH0=0x00;    //给定初值，这里使用定时器最大值从 0 开始计数一直到 65535 溢出
    TL0=0x00;
    EA=1;            //总中断打开
    ET0=1;           //定时器中断打开
```

```
        TR0=1;          //定时器开关打开
    }
```

2.代码 TMOD |= 0x01 程序中的 TMOD 为定时器的特殊功能寄存器，低半字节用于控制 T0 和高半字节用于控制 T1 的工作模式，51 单片机复位时，TMOD 所有位被清 0，请看下面的寄存器功能图图 3-43：

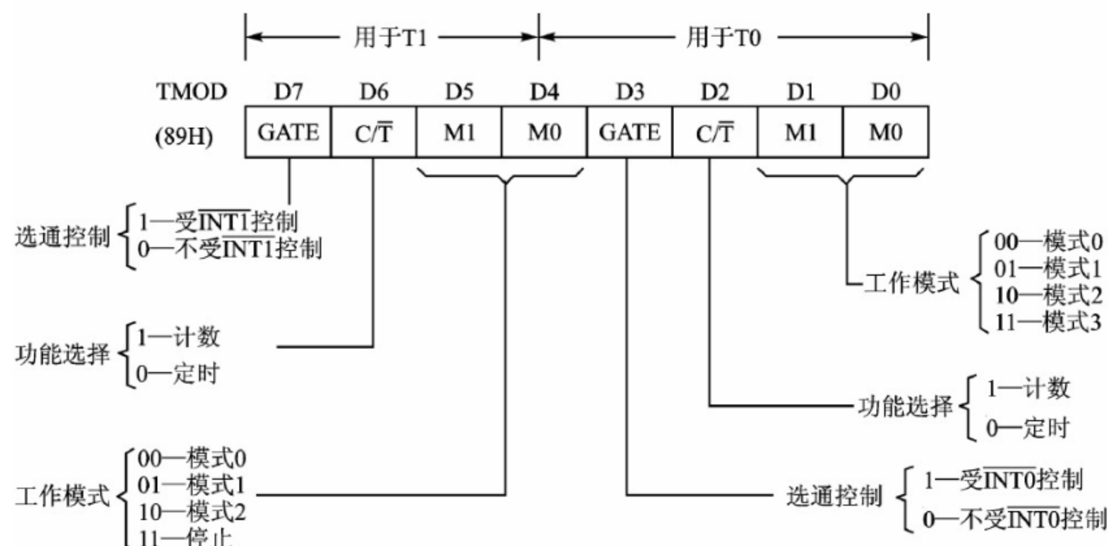


图 3-43 TMOD 寄存器功能

TMOD 的高 4 位用于设置定时器 1，低 4 位用于设置定时器 0，对应 4 位的含义如下：

GATE—门控制位；GATE=0，定时器/计数器启动与停止仅受 TCON 寄存器中 TR0、TR1 来控制；GATE=1，定时器/计数器启动与停止由 TCON 寄存器中 TR0、TR1 和外部中断引脚（INT0 或 INT1）上的电平状态来共同控制。程序中对 GATE 被设置为 0。

C/T—定时器模式和计数器模式选择位；C/T=1，为计数器模式； C/T=0，为定时器模式；程序里也是设置为 0。

M1 M0—工作方式选择位。这里 M1 被设置为 1，M0 被设置为 0；每个定时器/计数器都有 4 种工作方式，它们由 M0 M1 设定，对应关系表如下表 3-17 所示：

表 3-17 定时器/计数器的 4 种工作方式

M1	M0	工作方式
0	0	方式 0， 为 13 位定时器/计数器
0	1	方式 1， 为 16 位定时器/计数器
1	0	方式 2， 8 位初值自动重装的 8 为定时器/计数器
1	1	方式 3， 仅适用于 T0，分成两个 8 位计数器，T1 停止计数

代码 TMOD |= 0x01 使得单片机按定时器 0 方式 1 进行运行。

3.TH0 和 TL0 = 0x0 是定时器 0 的初值，定时器一旦启动，它便在原来的数值上开始加 1 计数，若在程序开始时，我们没有设置 TH0 和 TL0，它们的默认值都是 0，假设时钟频率为 12MHz，12 个时钟周期为一个机器周期，那么此时机器周期就是 1us，计满 TH0 和 TL0 就需要 2 的 16 次方-1 个数（因为方式 1 为 16 位的定时器/计数器），再来一个脉冲计数器溢出，随即向 CPU 申请中断。因此溢出一次共需 65536us，约等于 65.5ms，如果我们要定时 50ms 的话，那么就需要先给 TH0 和 TL0 装一个初值，在这个初值的基础上计 50000 个数后，定时器溢出，此时刚好就是 50ms 中断一次，当需定时 1s 时，我们写程序时当产生 20 次 50ms 的定时器中断后便认为是 1s，这样便可精确控制时间了。

当要计 30000 个数时，TH0 和 TL0 中应该装入的总数是 65536-30000=35536，把 35536 对 256 求模：35536/256=138 装入 TH0 中，把 35536 对 256 求余，35536%256=208 装入 TL0 中。

以上就是定时器的初值的计算方法，总结后得出如下结论：当用定时器的方式 1 时，设机器周期为 T_{cy}，定时器产生一次中断的时间为 t，那么需要计数的个数 N=t/T_{cy}，装入 THX 和 TLX 中的数分别为

$$THx=(65536-N)/256 \quad , \quad TLx=(65536-N)\%256$$

要计算机器周期 T_{cy}，就需要知道系统时钟频率，也就是单片机的晶振频率，TX-1C 实验板上时钟频率为 11.0592MHz，那么机器周期为 12×（1/11059200）≈1.09us， 若 t=50ms，那么 N=50000/1.09≈45872，这是晶振在 11.0592MHz 下定时 50ms 时初值的计算方法，当晶振为 12MHz 时，计算起来就比较方便了，用同样方法可算得 N=50000.

4.TR0=1 是将定时器开关打开，它是属于 TCON 寄存器的中的一个位，可以从<reg52.h>的头文件中看到：

```
/*   TCON   */
sbit TF1     = TCON^7;
sbit TR1     = TCON^6;
sbit TF0     = TCON^5;
sbit TR0     = TCON^4;
sbit IE1     = TCON^3;
sbit IT1     = TCON^2;
sbit IE0     = TCON^1;
sbit IT0     = TCON^0;
```

下面来看看 TCON 寄存器的各个字段的意思，如下图 3-44。

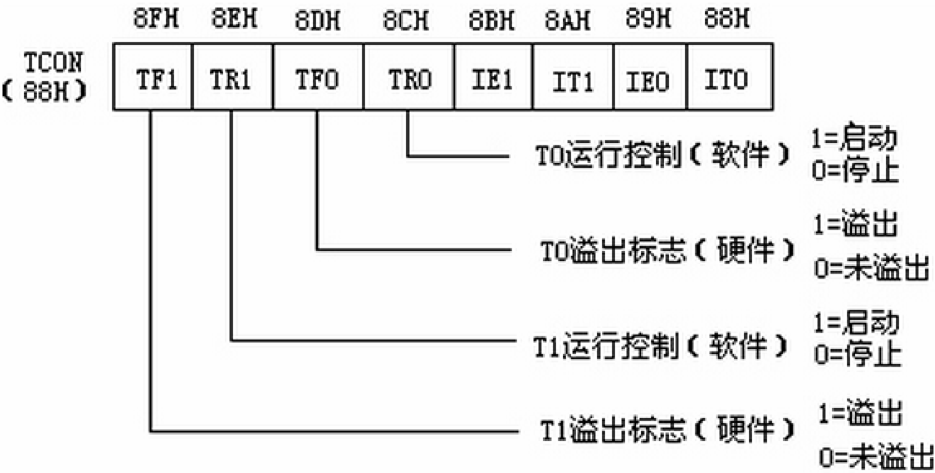


图 3-44 TCON 寄存器功能图

TCON 在特殊功能寄存器中，字节地址为 88H，由于有位地址，十分便于进行位操作。TCON 的作用是控制定时器的启、停，标志定时器溢出和中断情况。下面我们来了解下它的各位的定义：

TF1：定时器 1 溢出标志位。当定时器 1 计满溢出时，由硬件使 TF1 置“1”，并且申请中断。进入中断服务程序后，由硬件自动清“0”，在查询方式下用软件清“0”。

TR1：定时器 1 运行控制位。由软件清“0”关闭定时器 1。当 GATE=1，且 INT1 为高电平时，
TR1：置“1”启动定时器 1；当 GATE=0，TR1 置“1”启动定时器 1。
TF0：定时器 0 溢出标志。其功能及操作情况同 TF1。
TR0：定时器 0 运行控制位。其功能及操作情况同 TR1。
IE1：外部中断 1 请求标志位。
IT1：外部中断 1 触发方式选择位。当 IT1=0，为低电平触发方式；当 IT1=1，为下降沿触发方式。
IE0：外部中断 0 请求标志位。
IT0：外部中断 0 触发方式选择位。 当 IT0=0，为低电平触发方式；当 IT0=1，为下降沿触发方式。

6. EA=1 是总中断打开，允许中断的产生；代码 ET0=1 是定时器溢出时产生的中断被响应。更详细的中断内容可以参考对应的中断章节，里面会有详细的分析。

因为我们使用到了中断，所以还需要去了解一个寄存器，它就是中断允许寄存器 IE，如下表 3-18。

表 3-18 IE 中断允许寄存器

EA			ES	ET1	EX1	ET0	EX0
总中断			串行中断	T1 中断	外部中断 1	T0 中断	外部中断 0

EA 位为 1 时，允许中断，需要使用什么中断源在相对应的位置 1 即可。我们看到程序中是 EA=1 打开总中断，然后 ET0=1 打开定时器 0 的中断，这样的话，中断就打开了。TR 是定时计数器的启动控制开关；TR0=1：定时器开始计时； TR0=0：定时器停止计时。

中断函数通过使用 interrupt 关键字和中断编号 0-4 来实现。
使用该扩展属性的函数声明语法如下：
{ 返回值 函数名 interrupt n} n 对应中断源的编号

中断编号告诉编译器中断程序的入口地址，它对应者 IE 寄存器中的使能位，即 IE 寄存器中的 0 位对应着的外部中断 0，相应的外部中断 0 的中断编号是 0，中断编号的介绍如下表 3-19 所示

表 3-19 中断编号的说明

中断编号	中断源	入口地址
0	外部中断 0	0003H
1	定时器/计数器 0 溢出	000BH
2	外部中断 1	0013H
3	定时器/计数器 1 溢出	001BH
4	串行口中断	0023H

关键字 interrupt m [using n] 表示这是一个中断函数

m 为中断源的编号，有五个中断源，取值为 0,1,2,3,4，中断编号会告诉编译器中断程序的入口地址，执行该程序时，这个地址会传个程序计数器 PC，于是 CPU 开始从这里一条一条的执行程序指令。

n 为单片机工作寄存器组（又称通用寄存器组）编号，共四组，取值为 0,1,2,3。那么 using n 的意思是什么呢？在正在执行一个特定任务时，有更紧急的事情需要 CPU 来处理，

涉及到中断优先权。高优先权中断低优先权正在处理的程序，所以最好给每个优先程序分配不同的寄存器组。

CPU 正在处理某个事件，突然另外一个事件需要处理，于是进入中断后，而你不想将现在执行的程序的各寄存器状态入栈，那么可以把这个中断程序放入另一个寄存器组，如切换到 1 组，然后退出中断时，再切回到 0 组（原来的程序在 0 组）。

在中断函数中 void Timer0(void) interrupt 1 using 1

我们将对定时器重新初始化赋值，然后再将 LED 取反，如果是亮的，就让其灭掉，如果是灭的，就将其点亮；所以我们的程序将不断的执行中断函数，使得 LED 亮灭亮灭。

因为我们开启了中断 1，在 51 单片机内部，定时器 0 溢出就会使得中断响应，而中断响应的函数靠这句话来声明:

```
/*- 定时器中断子程序 -*/
Void Timer0(void) interrupt 1 using 1
```

其中 interrupt 1 表示中断 1，也就是将函数 Timer0()与中断 1 绑定起来，当发生中断 1 的时候，就会执行 Timer0()这个函数。

最后面的“using 工作组”是指这个中断函数使用单片机内存中 4 组工作寄存器中的哪一组，C51 编译器在编译程序时会自动分配工作组，因此最后这句话我们通常可以省略不写，因为如果你自己不会分配，有可能还会导致工作组资源冲突，还不如让编译器自动去分配的好。

```
LED = ~LED;     //指示灯反相，可以看到闪烁
```

所以当定时器时间到产生中断的时候 LED 就会取反，亮的，就让其灭掉，如果是灭的，就将其点亮，然后重新赋值，等到时间到产生中断后再次取反，所以我们就能看到 LED 按一定的时间进行闪烁了。

3.5.7 更多有关定时器例程

更多定时器相应的例程可通过北京航空航天大学出版社下载专区下载，如下表 3-20：

表 3-20 定时器更多丰富例程（含详细注释和文档分析）

序号	例程功能
例程 01	用定时器使得 LED 灯闪烁
例程 02	用定时器 1 使得 LED 灯闪烁
例程 03	用定时器 0 产生 10ms 的方波
例程 04	用定时器 0 产生 1s 的方波
例程 05	产生多路不同频率的方波

3.6 外部中断

3.6.1 什么是中断？

单片机中断系统的概念：什么是中断，我们从一个生活中的例程引入。比如说你在做 A

事，但是突然间来了你想起来了更重要的 B 事，所以你马上去做 B 事了，做完之后再回来继续做 A 事，这个就是中断。

3.6.2 什么是单片机的中断？

当 CPU 正在执行一个任务，但突然又发生了一个更高级的任务，CPU 必须立即去执行的任务，所以 CPU 必须中断当前的任务，并保存该任务已经执行的状态和相关信息，然后转而去执行那个更加高级的任务，因此就引入了“中断”这个概念。

中断是指计算机在执行程序的过程中，当出现异常情况或特殊请求时，计算机停止现行程序的运行，转向对这些异常情况或特殊请求的处理，处理结束后再返回现行程序的间断处，继续执行原程序。 中断是单片机实时地处理内部或外部事件的一种内部机制。当某种内部或外部事件发生时，单片机的中断系统将迫使 CPU 暂停正在执行的程序，转而去进行中断事件的处理，中断处理完毕后，又返回被中断的程序处，继续执行下去。

在程序里面也是一样的。举个例子可能会容易懂点，定时中断：比如你定时 1ms，主程序在运行，每当 1ms 时间到后，就跑到定时中断子程序里面执行，执行完后再回到主程序（中断程序是 1ms 中断一次）。那对于整个系统来说中断能实现什么好处呢？下面我们给以说明：

1）提高了 CPU 的效率

CPU 是计算机的指挥中心，它与外围设备（如按键、显示器等）通讯的方法有查询和中断 2 种： 查询的方法是无论外围 IO 是否需要服务，CPU 每隔一段时间都要依次查询一遍，这种方法 CPU 需要花费一些时间在做查询服务工作。

中断则是在外围设备需要通讯服务时主动告诉 CPU，这个时候 CPU 才停下当前工作去处理中断程序，不需要占用 CPU 主动去查询的时间，CPU 可以在没有中断请求来临之前一直做自己的工作，从而提高了 CPU 效率。

2）可以实现实时处理

外设任何时刻都可能发出请求中断信号，CPU 接到请求后及时处理，以满足实时系统的需要。

3）可以及时处理故障

计算机系统运行过程中难免会出现故障，有许多事情是无法预料的，如电源掉电、存储器出错、外围设备工作不正常等，这时可以通过中断系统向中断源 CPU 发送中断请求，由 CPU 及时转到相应的出错处理程序，从而提高计算机的可靠性。

3.6.3 什么是中断的来源

一个石头扔水里产生波纹，那么这个石头就是一个来源；同理，什么是中断的来源？什么可以引起中断？

生活中很多事件能引起中断：有人按了门铃了，电话铃响了，你的闹钟闹响了，你烧的水开了….等等诸如此类的事件。我们把能引起中断的称之为中断源，单片机中也有一些能引起中断的事件，51 单片机中一共有 5 个中断源，其中有两个外部中断和两个定时/计数器中断，再加一个串行口中断：

- 中断源 1：外部中断 0（INT0）
- 中断源 2：外部中断 1（INT1）

- 中断源 3：定时/计数器中断 0（T0）
- 中断源 4：定时/计数器中断 1(T1)
- 中断源 5：串行口中断

3.6.4 什么是中断的优先级

设想一下，我们正在看书，电话铃响了，同时又有人按了门铃，你该先做那样呢？这是两个中断同时到达的情况；假设我们正在看书，电话铃响了，我们去接听电话，接听到一半的时候，有人按门铃，这时候该怎么做呢？这是一个中断已经发生，又有一个中断产生的情况。这就涉及到一个优先级的问题了，假如看书的优先级>有人按门铃的优先级>电话铃响的优先级；那么我们应该一直看书，等看完书后，如果有人按门铃，我们再去开门，这个过程中，无论电话铃声是否响起，都不会去接，因为还有两个更高优先级的任务还没执行完。

在这里你一定会觉得单片机很笨是不，为什么不能同时做好这 3 件事情？但事实就是如此，除非你调高某个中断的优先级，你觉得重要就调高它，世界上没有完全完美的事务的。

言归正传，51 单片机中有一个专门设置中断优先级的寄存器，寄存器名称叫 IP 寄存器，在这个寄存器里已经默认了中断优先级：

外部中断 0 > 定时/计数器 0 > 外部中断 1 > 定时/计数器 1 > 串行中断。

这种优先级被称为逻辑优先级。这个时候，两个或两个以上的中断同时到达时，高的的优先级先得到服务。这种优先级实际上是中断同时到达的情况下，谁先得到服务的优先级，而不是可提供中断嵌套能力的优先级。例如：当一个中断 A 已经发生并执行着，这时候又有一个中断 B 发生，即使中断 B 的优先级高于 A 的优先级，也打断不了中断 A 的执行。

3.6.5 单个中断的响应过程

当有事件产生，进入中断之前我们必须先记住现在看书的第几页了，或拿一个书签放在当前页的位置，然后去处理不一样的事情（因为处理完了，我们还要回来继续看书）：电话铃响我们要到放电话的地方去，门铃响我们要到门那边去，也说是不一样的中断，我们要在不一样的地点处理，而这个地点常常还是固定的。51 单片机中也是采用的这种办法，五个中断源，每个中断产生后都到一个固定的地方去找处理这个中断的程序，当然在去之前首先要保存下面将执行的指令的地址，以便处理完中断后回到原来的地方继续往下执行程序。

单片机响应中断的流程图如下图 3-45：

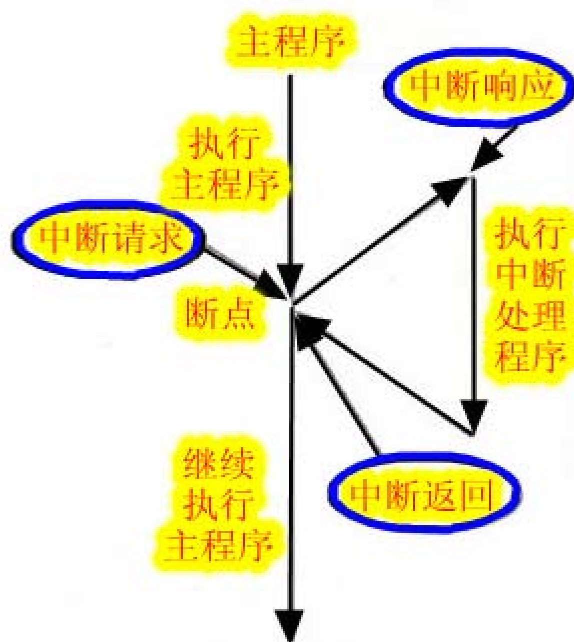


图 3-45 中断流程图

中断的处理过程：

1、 中断当前程序并保护断点
2、 转入中断服务入口
3、 保护现场
4、 执行中断服务程序
5、 恢复现场
6、 中断返回

具体地说，中断响应能分为以下几个步骤：

A、保护断点，即保存下一将要执行的指令的地址，就是把这个地址送入堆栈。

B、寻找中断入口，根据 5 个不一样的中断源所产生的中断，查找 5 个不一样的入口地址。以上工作是由计算机自动完成的，与编程者无关。在这 5 个入口地址处存放有中断处理程序（这是程序编写时放在那儿的，如果没把中断程序放在那儿，就错了，中断程序就不能被执行到）。

C、执行中断处理程序。

D、中断返回：执行完中断指令后，就从中断处返回到主程序，继续执行。究竟单片机是怎么样找到中断程序所在位置，又怎么返回的呢？我们稍后再谈。

3.6.6 多个中断的嵌套响应过程

中断的嵌套可以这样理解：程序正在执行，发生中断 A，CPU 响应执行中断 A；这时发生中断 B（B 能打断 A，实现中断嵌套），CPU 则响应执行中断 B。B 中断处理完成后，返回处理中断 A，A 中断处理完后，再发回主程序继续执行。

单片机中断嵌套流程图如下图 3-46：

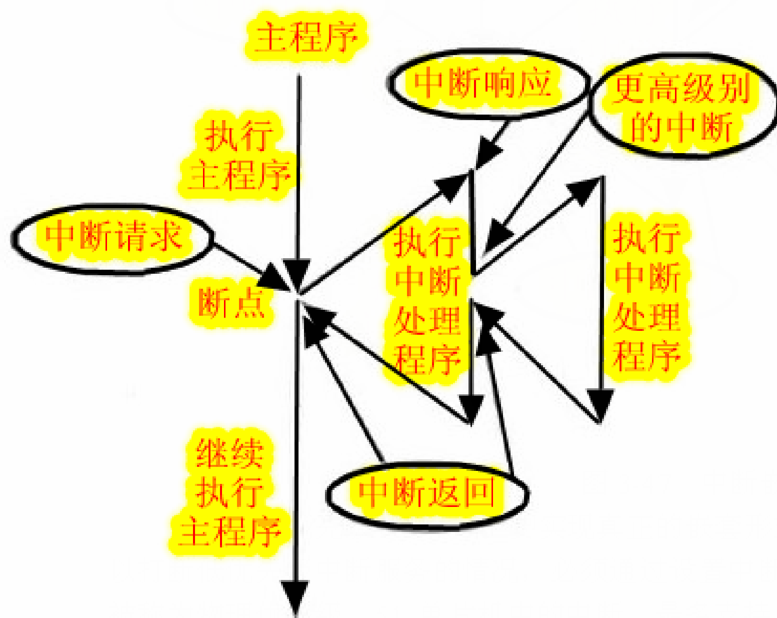


图 3-46　中断嵌套流程图

设置中断优先级寄存器 IP 要实现真正的嵌套形式的优先级，也即高优先级中断服务可以打断低优先级中断服务的情况，必须通过设置中断优先级寄存器 IP 来实现；这种优先级被称为物理优先级。51 单片机中的中断，最多支持二级嵌套。例如：我通过设置中断优先级寄存器，设置中断 A 为最高优先级；则中断 A 可以打断任何其他的中断服务函数实现嵌套，且只有中断 A 能打断其他中断的服务函数。若中断 A 没有触发，则其他中断之间还是保持逻辑优先级，相互之间无法嵌套。

3.6.7 单片机中的中断如何被管理

51 单片机里总共含有 5 个中断源，分别是 2 个外部中断，2 个定时器中断和 1 个串口中断；可以从下表看到，这 5 个中断，除了外部中断 0 和外部中断 1 是由两个 51 单片机管脚 P3.2 和 P3.3 提供之外，其他 3 个中断都是内部响应和提供，也就是说由 51 单片机内部本身提供，如下表 3-21：

表 3-21 中断源的介绍

中断名称	中断源
外部中断 0	外部中断由 P3.2 提供
外部中断 1	外部中断由 P3.3 提供
定时器 0 溢出中断	由片内定时器/计数器 0 提供
定时器 1 溢出中断	由片内定时器/计数器 1 提供
串口中断	由片内串口提供

这 5 个中断源是被 4 个寄存器管理着的，这 4 个寄存器分别是控制中断屏蔽或使能的 IE 中断允许控制寄存器，管理各个中断优先级的 IP 中断优先级控制寄存器、还有另外两个定时器/计数器中断寄存器以及串口中断寄存器等，分别控制定时器的状态和串口的收发数据情况，控制什么时候响应中断，进入中断处理函数。

下表分别介绍 4 个寄存器，单片机的 5 个中断就是由这几个寄存器各司其职来进行控制

的：

表 3-22 IP 中断优先级控制寄存器

位序号	D7	D6	D5	D4	D3	D2	D1	D0
位符号	--	--	--	PS	PT1	PX1	PT0	PX0
位地址	--	--	--	BCH	BBH	BAH	B9H	B8H

中断是分优先级的，当几个中断事件同时来的时候，我们可以根据轻重缓急来命令 CPU 优先执行哪个中断，那么这个轻重缓急就是我们通过这个寄存器进行设置。

PS：串行口优先级设定位。

PT0：定时器中断 0 优先级设定位

PT1：定时器中断 1 优先级设定位

PX0：外部中断 0 优先级设定位

PX1：外部中断 1 优先级设定位

为“0”时优先级为低，为“1”时优先级为高。51 单片机只提供二级中断嵌套。

表 3-23 SCON 串口控制寄存器

位序号	D7	D6	D5	D4	D3	D2	D1	D0
位符号	SM0	SM1	SM2	REN	TB8	RB8	TI	RI
位地址	9FH	9EH	9DH	9CH	9BH	9AH	99H	98H

这是一个专门针对串口的寄存器，它的主要功能是串口发一段或接受一段数据，要做一个标志，通知 CPU 去处理完，再发接下来的数据，这样就可以保障数据不会因为上批数据还未读完，新的数据又来了，把旧数据冲刷掉。

3.6.8 硬件原理说明

我们将单片机的 P3.2 管脚与独立键盘的一个按键连接，使得我们在按下按键的时候，改变 P3.2 管脚的电平触发中断的产生。

3.6.9 例程 01 外部中断 0 电平触发

代码如下：

```
/**************************************************
*名称：外部中断 0 电平触发
*作者：www.armjishu.com
*版本：v1.0
*内容：通过中断接口 P3.2 连接的独立按键 K1 测试，按一次 P2 口的 LED 灯反向，这里使用电平触发，
        所以一直按键不松开和一次按键效果不相同，按下会看到灯全部亮说明中断一直在作用
**************************************************/
#include<reg52.h>   //包含头文件，一般情况不需要改动，
                    //头文件包含特殊功能寄存器的定义
/*-- 主程序 --*/
main()
{
```

```
        P2=0x0f;            //P2 口初始值,4 盏灯亮，4 盏灯灭
        EA=1;               //全局中断开
        EX0=1;              //外部中断 0 开
        IT0=0;              //电平触发
        while(1)
        {
                          //在此添加其他程序
        }
    }
    /*-- 外部中断程序 --*/
    void ISR_Key(void) interrupt 0 using 1
    {
        P2 = ~P2;          //进入中断程序执行程序，
                         //此时可以通过 EA=0 指令暂时关掉中断
    }
```

硬件连接如表 3-24

表 3-24 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P2 口	JP16	直连	JP19	1 根 8 针扁平电缆	控制 LED 灯
P3.2 管脚	JP14	直连	JP18	1 根 1 针杜邦线	控制独立按键
实验现象：下载程序后，连好杜邦线，按下按键，P2 口连的 LED 灯取反（执行一次中断 LED 灯就取反一次）；如果一直按下，LED 灯就全部点亮，表示中断一直进行，不间断的取反操作，所以看不到变化					

知识要点：

1. 首先，根据下表 3-24 中断源的介绍，外部中断 0 就是由 P3.2 管脚来负责的，所以在硬件原理图上要连接好 P3.2 管

表 3-25 中断源介绍

中断名称	中断源
外部中断 0	外部中断由 P3.2 提供
外部中断 1	外部中断由 P3.3 提供
定时器 0 溢出中断	由片内定时器/计数器 0 提供
定时器 1 溢出中断	由片内定时器/计数器 1 提供
串口中断	由片内串口提供

2. 该语句“void ISR_Key(void) interrupt 0 using 1”设置中断 0，并且在这个函数里对 P2 所连的 8 个 LED 灯进行取反。

------ ISR_Key 这个名字可以随便写，中断函数只与中断号绑定

------ 中断号是什么? Interrupt 0 就是表示中断号 0，只要有外部中断响应，就会调用 ISR_Key()这个函数，因为“void ISR_Key(void) interrupt 0“这句代码将中断 0 与 ISR_Key()这个函数绑定起来了，请见下表 3-26：

表 3-26 中断号的介绍

中断名称	中断号
外部中断 0	中断号 0
定时/计数器 0	中断号 1

外部中断 1	中断号 2
定时/计数器 1	中断号 3
串口中断	中断号 4
定时/计数器 2	中断号 5

------该语句“void ISR_Key(void) interrupt 0 using 1”中的 using 1 是什么意思呢？using 后面的数字一般都是从 0~3 都可以，因为 51 单片机中有 4 组寄存器，选择了其中一组相当于当中断来的时候，51 单片机就把目前的状态情况保存到该组寄存器中去，在这里推荐初学者朋友不要加这句话上去，因为怕这几组寄存器设定其他中断的时候，冲突了；如果不手动加上去，编译器也会自动去分配的，自动分配就不会有冲突这个问题出现，所以实际上最好的代码的是
“void ISR_Key(void) interrupt 0“这样就可以了，这句代码等同于“void ISR_Key(void) interrupt 0 using 1”除非对这个部分非常了解和精通的朋友除外。

3．在主函数中，按照中断初始化的流程，首先代码“EA=1”打开全局的总中断，然后外部中断 0 打开“EX0=1”；关于 EA 可以从‘**IE 中断允许控制寄存器**’表 3-27 中查到，下面是 IE 中寄存器中各个 BIT 的值和所代表的意思：

表 3-27 IE 寄存器 BIT 值介绍

位序号	D7	D6	D5	D4	D3	D2	D1	D0
位符号	EA	--	--	ES	ET1	EX1	ET0	EX0
位地址	AFH	AEH	ADH	ACH	ABH	AAH	A9H	A8H

主要任务是控制中断是禁止还是允许，相当于一个中断的总开启的开关，只有开启了这个开关，中断才会有效；因为中断是会打扰和影响到 CPU 的工作的，所以有必要为此设定一个开关，这样可以最大使得 CPU 的效率最高。

------第 7 位：EA： 全局中断控制位（EA=1 打开，EA=0 关闭）
在 EA=1 的条件下，由各个中断控制位确定相应中断的打开或关闭。

-------第 6 位和第 5 位为空

------第 4 位：ES： 串行口中断控制位（ES=1 充许，ES=0 禁止）

-------第 3 位和第 1 位：ET0 和 ET1： 定时/计数器中断控制位（ET0(ET1)=1 允许， ET0(ET1)=0 禁止）。分别控制两个片内定时/计数器 T0 和 T1。

------第 2 位和第 0 位:EX0 和 EX1： 外部中断控制位(EX0(EX1)=1 允许,EX0(EX1)=0 禁止）分别控制外部中断 INT0 和 INT1。

那么关于 EA=1 这句代码可以打开<reg52.h>这个文件查看到 “sbit EA = IE^7”,也就是 IE 寄存器的第 7 位

```
/*  IE  */
sbit EA    = IE^7;
sbit ET2   = IE^5; //8052 only
sbit ES    = IE^4;
sbit ET1   = IE^3;
sbit EX1   = IE^2;
sbit ET0   = IE^1;
sbit EX0   = IE^0;
```

继续在<reg52.h>这个文件里查看可以看到 `sfr IE = 0xA8;` ,可以看到这个，刚好

与这个表里的位地址的最低位相同，这样代码就与手册里的说明地址对应上了，实际上 EA 的位序号是 D7，位地址是 0xAF。如下表 3-28

表 3-28 IE 寄存器 BIT 值介绍

位序号	D7	D6	D5	D4	D3	D2	D1	D0
位符号	EA	--	--	ES	ET1	EX1	ET0	EX0
位地址	AFH	AEH	ADH	ACH	ABH	AAH	A9H	A8H

4.接下的代码 IT0=0 这句代码可以从 TCON 寄存器里看到 IT0 这个位的身影，寄存器的的选项 IT0=0 是电平触发，IT1=1 是边沿下降沿触发，下面我们对这个寄存器做一个详细的说明，如下表 3-29：

表 3-29 TCON（Timer Control Register）定时器/计数器控制寄存器

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

- TF1：定时器 1 溢出标志位。当字时器 1 计满溢出时，由硬件使 TF1 置“1”，并且申请中断。进入中断服务程序后，由硬件自动清“0”，在查询方式下用软件清“0”
- TR1：定时器 1 运行控制位。由软件清“0”关闭定时器 1。当 GATE=1，且 INT1 为高电平时，TR1 置“1”启动定时器 1；当 GATE=0，TR1 置“1”启动定时器 1。
- TF0：定时器 0 溢出标志。其功能及操作情况同 TF1。
- TR0：定时器 0 运行控制位。其功能及操作情况同 TR1。
- IE1：外部中断 1 请求标志。
- IT1：外部中断 1 触发方式选择位，其中 IT0=0 是电平触发，IT1=1 是边沿下降沿触发。
- IE0：外部中断 0 请求标志。

3.6.10 更多有关外部中断例程

更多外部中断相应的例程可通过北京航空航天大学出版社下载专区下载，如下表 3-30：

表 3-30 外部中断更多丰富例程介绍（含详细注释和文档分析）

序号	例程功能
例程 01	外部中断 0 电平触发
例程 02	外部中断 1 电平触发
例程 03	外部中断 0 边沿触发
例程 04	外部中断 1 边沿触发
例程 05	计数器 T0 外部中断输入
例程 06	计数器 T1 外部中断输入

3.7 蜂鸣器（喇叭）

3.7.1 蜂鸣器的简介

蜂鸣器是一种一体化结构的电子讯响器，采用直流电压供电，广泛应用于计算机、打印机、复印机、报警器、电子玩具、汽车电子设备、电话机、定时器等电子产品中用作发声器

件。

3.7.2 蜂鸣器深入分析

蜂鸣器由振荡器、电磁线圈、磁铁、振动膜片及外壳等组成，声音主要是靠振动膜片的周期振动发生，而振动膜片的振动是靠电子线圈磁场产生动能来驱动的；即接通电源后，外接的振荡器产生的音频信号电流通过电磁线圈，使电磁线圈产生磁场，振动膜片在电磁线圈和磁铁的相互作用下，周期性地振动发声。

一般的无源蜂鸣器的结构图，如下图 3-47：

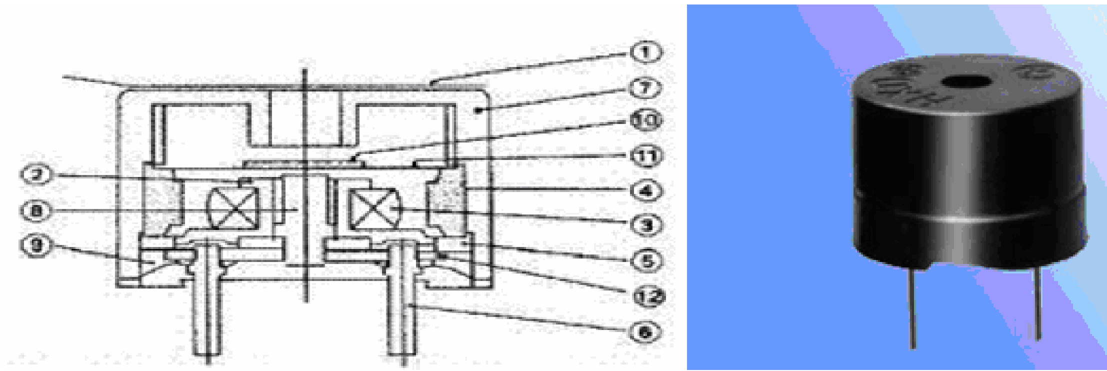


图 3-47 无源蜂鸣器结构图

1：防水贴纸　　7：外壳
2：线轴　　8：铁蕊
3：线圈　　9：封胶
4：磁铁　　10：小铁片
5：底座　　11：振动膜
6：引脚　　12：电路板

因此要使得这种蜂鸣器发出声音，必须在外部给它接一个震荡发生器，比如单片机内部的 PWM 波。

蜂鸣器可以分为电磁式和电压式或者有源蜂鸣器和无源蜂鸣器，这里讲的“源”，指的是振荡源；有源蜂鸣器，内部有振荡、驱动电路，加电源就可以响，优点是用起来省事；缺点是频率固定了，就只一个单音；无源的蜂鸣器与喇叭一样，需要加上交变的音频电压才能发声，也可以发出不同频率的声音。不过，蜂鸣器的声音反正是不好听的，所以经常是加上方波，而不是加正弦。

电压式蜂鸣器主要由多谐振荡器、压电蜂鸣片、阻抗匹配器及共鸣箱、外壳等组成。它是以压电陶瓷的压电效应，来带动金属片的振动而发声的；电磁式的蜂鸣器，则是用电磁的原理，通电时将金属振动膜吸下，不通电时依振动膜的弹力弹回，故压电式蜂鸣器是以方波来驱动。

3.7.3 蜂鸣器和喇叭的区别

首先可以认为蜂鸣器是一种最简单的喇叭，通电之后发声，按照能量转换原理，就是电能(蜂鸣器通电)→磁能(导电的线圈产生磁能)→动能(薄膜振动)→声能(发声)，这就是蜂鸣器发声的原理，可以看到电流导致磁场的产生并导致振动而使得薄膜发声。

喇叭实际上由两个磁场交互的结果，请注意喇叭多了一对磁铁，而永久磁铁产生一个大小与方向不变的恒定磁场，而此时音圈产生一个磁场大小和方向随音频电流的变化不断地在改变的磁场时，这样两个磁场的相互作用使音圈作垂直于音圈中电流方向的运动，由于音圈和振动膜相连，从而音圈带动振动膜振动，由振动膜振动引起空气的振动面发出声响。

当喇叭的音圈通入音频电流后，音圈在电流的作用下便产生了交变磁场，随着电流大小的变化，就可以实现发出不同的声音了。输出给音圈的电流越大，其磁场的作用力就越大.振动膜振动的幅度也就越大，声音则越响；喇叭发出高音的部分主要在振动膜的中央，喇叭发出低音的部分主要在振动膜的边缘。如果喇叭的振动膜边缘较为柔软且纸盆口径较大则喇叭发出的低音效果较好，所以评价一个喇叭的质量好坏就是基于这些原理来进行区分的，更加详细的内容可以进一步查找相关资料。

3.7.4 硬件原理与连接

硬件连接原理图如图 3-48 所示

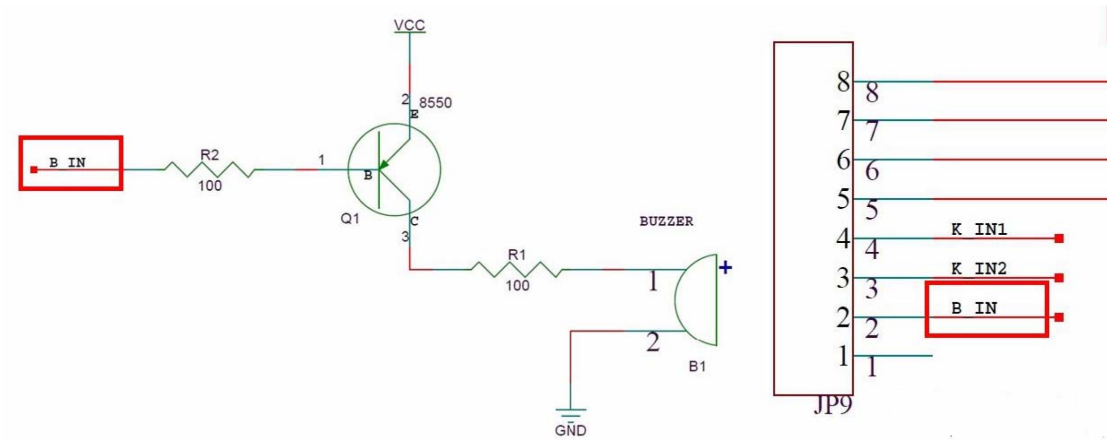


图 3-48 无源蜂鸣器硬件原理图

B_IN 引脚控制 Q1PNP 三极管的导通与截止。当 B_IN 引脚为高电平时，PNP 三极管导通，蜂鸣器正极为高电平。当 B_IN 引脚为低电平时，PNP 三极管截止，蜂鸣器正极为低电平。从而形成交变的音频电压，使蜂鸣器发声。

3.7.4 例程 01 喇叭发声原理

代码如下：

```
/**************************************************
*名称：喇叭
*作者：www.armjishu.com
*版本：v1.0
*内容：通过发出一定频率方波，是喇叭发声
**************************************************/
#include<reg52.h>    //包含头文件，一般情况不需要改动，
                     //头文件包含特殊功能寄存器的定义
sbit SPK=P1^2;       //定义喇叭端口
void Delay(unsigned char t)
```

```
{
    while(--t);
}
/*-- 主函数 --*/
main()
{
    while(1)
    {    /* 利用 SPK 一低一高模拟输出方波 */
        Delay(100);   //发出方波,频率越大声音越尖
        SPK=!SPK;
    }
}
```

硬件连接关系如表 3-31

表 3-31 无源蜂鸣器硬件连接关系

用排线电缆或杜邦线连接“单片机 IO”和“模块接口”					
单片机接口	插座 1	方式	插座 2	线缆	功能
P1.2 管脚	JP13	直连	JP9 的第 2 脚	1 根 1 针杜邦线	控制蜂鸣器
实验现象：下载程序后，连好杜邦线，蜂鸣器会发出叫声					

知识要点：

1.模拟出方波，使得开发板上的无源蜂鸣器发出响声，因为无源蜂鸣器需要加上交变的电压才能发生，所以通过 SPK=!SPK 和一个延时，保持一段时间高电压，又保持一段时间低电压，相当于一高一低一高一低，从而模拟出了方波

2.如果不模拟出方波可以使得蜂鸣器响吗？答案是有源蜂鸣器不需要方波，但是无源蜂鸣器需要，所以在该例程中，如果不模拟出方波，直接对蜂鸣器以单一的高电平或者低电平驱动是不会发声的。

3.7.5 更多蜂鸣器的例程

更多蜂鸣器相应的例程可通过北京航空航天大学出版社下载专区下载，如下表 3-32：

表 3-32 更多蜂鸣器丰富例程介绍（含详细注释和文档分析）

序号	例程功能
例程 01	喇叭发声原理
例程 02	模拟警车抓小偷的声音
例程 03	模拟传统电话机的声音
例程 04	喇叭嘀嗒声音
例程 05	报警发声
例程 06	模拟消防车声音
例程 07	音乐世上只有妈妈好

3.8 看门狗

3.8.1 什么是看门狗

在由单片机构成的微型计算机系统中，由于单片机的工作常常会受到来自外界电磁场的干扰，造成程序的跑飞，而陷入死循环，程序的正常运行被打断，由单片机控制的系统无法继续工作，会造成整个系统的陷入停滞状态，发生不可预料的后果，所以出于对单片机运行状态进行实时监测的考虑，便产生了一种专门用于监测单片机程序运行状态的芯片，俗称"看门狗"(watchdog)。

3.8.2 看门狗的原理

如果初次接触看门狗，可能感觉有点难理解。其实很简单，可以把看门狗理解为一个能装有限数值的计数器。就是当启动“看门狗”后，它就不停的数机械周期，每数一个机械周期就加一，当它装不下了就会溢出，从而产生一个复位信号，重新启动系统。

具体来说：看门狗，又叫 watchdog timer，是一个定时器电路，一般有一个输入，叫喂狗(kicking the dog or service the dog)，一个输出到 MCU 的 RST 端，MCU 正常工作的时候，每隔一段时间输出一个信号到喂狗端，给 WDT 清零，如果超过规定的时间不喂狗，（一般在程序跑飞时），WDT 定时超过，就会给出一个复位信号到 MCU，使 MCU 复位. 防止 MCU 死机. 看门狗的作用就是防止程序发生死循环，或者说程序跑飞。

3.8.3 采用哪种看门狗

有些芯片内部集成了看门狗功能，如果没有集成，就需要额外采购专门的看门狗芯片。

内部集成的看门狗有哪些利弊？假如芯片内置的“看门狗”功能，使用片内看门狗进行喂狗，如果这个芯片损坏那看门狗也跟着失效，并且看门狗无法将这个故障消息告诉更高一级的系统，这是一个非常大的弊端，因为添加看门狗主要就是为了监控芯片是不是在正常工作，如果芯片出现故障，看门狗也无法使用了那就失去了意义。

采用专门的看门狗芯片来设计产品有哪些利弊？为了提高系统的可靠性，可以考虑添加外部的“看门狗”芯片，当看门狗监控的芯片故障或者程序跑飞，没有按时喂狗，这时看门狗就会报警，及时采取复位等有效的恢复措施，这样就可以实现看门狗真正的功能，设计出高可靠性的产品，不过因为增加了额外的“看门狗”芯片，从而增加了成本。

在使用产品过程中，虽然使用了看门狗，但也不能 100%保证能每次都报警故障，因为有可能看门狗自己失效，所以要设计一款可靠性较高的产品，必须利用看门狗的特点来进行有效的设计搭配，比如设计两个看门狗等等。

3.8.4 自己动手设计一个看门狗

自己动手设计一个看门狗可以吗？当然可以，比如小时候爸妈下班回家为了监督儿子不看电视，回家之后会摸一下电视机后面是否发热，如果发热表示儿子刚看过电视不久，儿子

挨揍；如果电视不发热，则证明儿子在认真学习，没有看电视；这个思路就是建立一个标志，通过查看这个标志来从而判断儿子是不是在认真学习，当然这个例子非常有局限性，只是说明一下这个原理。

现在用 51 单片机里的两个定时器来设计一套具有看门狗功能的程序来，具体思路如下：首先可以对 T0 设定一定的定时时间，当产生定时中断的时候对一个变量进行赋值，而这个变量在主程序运行的开始已经有了一个初值，在这里我们要设定的定时值要小于主程序的运行时间，这样在主程序的尾部对变量的值进行判断，如果值发生了预期的变化，就说明 T0 中断正常，如果没有发生变化则使程序复位。对于 T1 我们用来监控主程序的运行，我们给 T1 设定一定的定时时间，在主程序中对其进行复位，如果不能在一定的时间里对其进行复位，T1 的定时中断就会使单片机复位。在这里 T1 的定时时间要设的大于主程序的运行时间，给主程序留有一定的的余量。而 T1 的中断正常与否我们再由 T0 定时中断子程序来监视。这样就够成了一个循环，T0 监视 T1，T1 监视主程序，主程序又来监视 T0，从而保证系统的稳定运行。

真正的看门狗，实现了上面的这些功能，不需要那么麻烦，只需要写和读相关的看门狗寄存器就可以实现上面这些复杂的功能，看门狗自己会定时去喂狗，具体可以学习下面的实际例程，留意看门狗控制寄存器 WDT_CONTR 设置。

3.8.3 例程 01 看门狗溢出复位实验

代码如下：

```
/***********************************************************
* 例程：看门狗溢出复位实验
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：通过按键阻止喂狗模拟看门狗溢出
* 现象：下载程序后，所有 LED 被点亮一小段时间（大约 0.5S），所有指示灯亮为 0xAA。
        当连接到 P3.0 的按键持续按下时，指示灯亮为 0x55，一段时间以后看门狗会
溢出导致复位，
        复位后所有 LED 都会被点亮一次。
        如果按键持续按下，则重复上述过程，LED 一会儿全亮，一会而亮为 0x55，
导致闪烁。
***********************************************************/
#include<reg52.h>
/* STC 单片机的看门狗寄存器 */
sfr WDT_CONTR = 0xE1;
/* 其他厂商单片机的看门狗寄存器 */
sfr WDTRST     = 0xA6;
sbit key = P3^0;
/*------------------------------------------------
                喂狗函数
------------------------------------------------*/
void Rst_Watchdog( void )
{
    /* STC 单片机的看门狗喂狗函数 */
```

```
    /* 0011,1100 EN_WDT = 1,CLR_WDT = 1,IDLE_WDT = 1,PS= 011 */
    WDT_CONTR = 0x3B;       // 0.5688 S@11.0592MHz

    /* 其他厂商单片机的看门狗喂狗函数 */
    //WDTRST   = 0x1E; //先赋值 1E 然后赋值 E1
    //WDTRST   = 0xE1;
}
/*-----------------------------------------------
                    主函数
-----------------------------------------------*/
void main( void )
{
    /* unsigned int 是定义无符号整形变量，其值的范围是 0~65535 */
    unsigned int i;
    /*复位后点亮所有 LED 并保持一段时间 */
    P1=0x00;
    for( i = 0; i < 30000; i++)
    {
        P1=0x00;
    }
    /* 设置使能看门狗 */
    Rst_Watchdog();
    while(1)
    {
        /* 喂狗: 重新设置看门狗 */
        Rst_Watchdog();

        /* LED 显示 0xAA */
        P1=0xAA;
        /* 如果按键长期按下不松开，则程序一直在此处循环           */
        /* 导致喂狗程序不能被调用，超过喂狗最大间隔将导致复位 */
        while(!key)
        {
            /* 按键按下时 LED 显示 0x55 */
            P1=0x55;
            /* 如果按键持续按下，复位后会再次进入此循环，导致 LED 闪烁 */
        }
    }
}
```

硬件连接关系如表 3-33

表 3-33 看门狗硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P1 口	JP13	直连	JP19	1 根 8 针扁平电缆	控制 LED 灯
P3.0	JP14.1	直连	JP18.1	1 根杜邦线	检测按键状态
实验现象：下载程序后，所有 LED 被点亮一小段时间（大约 0.5S），所有指示灯亮为 0xAA。 当连接到 P3.0 的按键持续按下时，指示灯亮为 0x55，一段时间以后看门狗会溢出导致复位，复位后所有 LED 都会被点亮一次。 如果按键持续按下，则重复上述过程，LED 一会儿全亮，一会而亮为 0x55，导致闪烁。					

知识要点：

1．本实验是通过按键阻止喂狗模拟看门狗溢出。按键一直按下，使得程序不能进入喂狗的状态，一直到看门狗溢出，造成程序复位。

2. 由于看门狗电路为单片机内部电路，所以对外部电路没有特殊要求。但试验中用到 LED 指示灯和按键来配合表现实验结果，所以单片机的 P1 也就是神舟开发板的 JP13 排针连接到 LED 指示灯的 JP19，将单片机的 P3.0 也就是神舟开发板的 JP14 的第一个插针，连接到独立按键 JP18 的第一个插针。

3. 8051 单片机看门狗定时器特殊功能寄存器 WDT_CONTR 如下表 3-34：

表 3-34 寄存器 WDT_CONTR 介绍

寄存器标识	地址	功能含义	7	6	5	4	3	2	1	0	复位值
WDT_CONTR	E1h	看门狗控制寄存器	-	-	EN_WDT	CLR_WDT	IDLE_WDT	PS2	PS1	PS0	xx000000

EN_WDT：看门狗允许位, 当设置为 “1”时，看门狗启动。

CLR_WDT：看门狗清“0”位, 当设为“1”时，看门狗将重新计数。硬件将自动清 0。

IDLE_WDT：看门狗“IDLE”模式位, 当设置为“1”时, 看门狗定时器在“空闲模式”计数，当清“0”该位时, 看门狗定时器在“空闲模式”时不计数。

PS2,PS1,PS0 看门狗定时器预分频值，如下表 3-35 所示：

表 3-35 看门狗定时器预分频值介绍

PS2	PS1	PS0	预分频系数	看门狗周期@20MHz晶振 12倍时钟模式
0	0	0	2	39.3 mS
0	0	1	4	78.6 mS
0	1	0	8	157.3 mS
0	1	1	16	314.6 mS
1	0	0	32	629.1 mS
1	0	1	64	1.25S
1	1	0	128	2.5S
1	1	1	256	5S

看门狗溢出时间计算:

看门狗溢出时间=（ N x 预分频系数 x 32768）/ 晶振频率。其中, N 是单片机的时钟周期，可以在烧写程序时修改。当在 12 clock mode 时 N = 12；当在 6 clock mode 时 N =

6。设时钟为 12MHz，12 倍时钟模式时，看门狗溢出时间 = （12 x 预分频系数 x 32768）/ 12000000 = 预分频系数 x 393216 / 12000000。如下表 3-36

表 3-36 看门狗定时器预分频值介绍

PS2	PS1	PS0	预分频系数	看门狗周期@12MHz晶振 12倍时钟模式
0	0	0	2	65.5 mS
0	0	1	4	131.0 mS
0	1	0	8	262.1 mS
0	1	1	16	524.2 mS
1	0	0	32	1.0485S
1	0	1	64	2.0971S
1	1	0	128	4.1943S
1	1	1	256	8.3886S

设时钟为11.0592MHz, 12倍时钟模式

看门狗溢出时间 = (12 x 预分频系数 x 32768) / 11059200 = 预分频系数 x 393216 / 11059200，如下表3-37：

表 3-37 看门狗定时器预分频值介绍

PS2	PS1	PS0	预分频系数	看门狗周期@11.0592MHz晶振 12倍时钟模式
0	0	0	2	71.1 mS
0	0	1	4	142.2 mS
0	1	0	8	284.4 mS
0	**1**	**1**	**16**	**568.8 mS（示例程序使用的看门狗周期）**
1	0	0	32	1.1377S
1	0	1	64	2.2755S
1	1	0	128	4.5511S
1	1	1	256	9.1022S

3.“Rst_Watchdog”函数中的“WDT_CONTR = 0x3B;”即为STC单片机的看门狗使能与喂狗的功能实现。如果使用的外部晶振频率为11.0592MHz，寄存器设为**0x3B时**，预分频系数为16，则看门狗溢出时间，也就是最大喂狗间隔为0.5688S。

寄存器设为**0x3B**二进制为0011,1100，表示EN_WDT = 1时能看门狗，CLR_WDT = 1看门狗将重新计数，IDLE_WDT = 1看门狗定时器在“空闲模式”依然计数，PS= 011根据上述外部晶振频率为11.0592MHz的表格可以查出看门狗溢出时间为0.5688 S。当然也可以按照公司计算：

看门狗溢出时间 = (12 x 预分频系数 x 32768) / 11059200 =预分频系数 x 393216 / 11059200 = 16x393216/11059200 = 0.5688 S

4.“main”函数一开始便后点亮所有 LED（P1=0x00），并延迟一段时间，这样便于依据指示灯的状态来判断是否发生了复位。所有 LED 被点亮意味着发生复位。然后调用“Rst_Watchdog”函数使能看门狗。之后便进入一个 while(1)死循环。如果连接到 P3.1 的按键没有被按下，则指示灯亮为 0xAA（化为二进制为 1010 1010，即 4 亮 4 灭），此时程序一直在喂狗，看门狗不会溢出。当连接到 P3.1 的按键按下时指示灯亮为 0x55。如果长按下按键不放，程序一直在 while(!key)中循环，无法喂狗，一段时间以后看门狗会溢出导致复位。是否发生可复位事件可以通过所有 LED 是否被点亮来判断，所有 LED 被点亮意味着发生复位。复位后点亮所有 LED 如果按键持续按下，则重复上述过程。

3.8.4 更多看门狗的例程

更多看门狗相应的例程可通过北京航空航天大学出版社下载专区下载，如下表 3-38：

表 3-38 更多看门狗丰富例程介绍（含详细注释和文档分析）

序号	例程功能
例程 01	看门狗溢出复位实验
例程 02	按键控制定时喂狗实验

3.9 红绿双色点阵

3.9.1 LED点阵简介

LED 点阵是由发光二极管排列组成的显示器件,在我们日常生活的电器中随处可见，被广泛应用于汽车报站器，广告屏等。特别是它的发光类型属于冷光源，效率及发热量是普通发光器件难以比拟的，它采用低电压扫描驱动，具有：耗电少、使用寿命长、成本低、亮度高、故障少、视角大、可视距离远、规格品、可靠耐用、应用灵活、安全、响应时间短、绿色环保、控制灵活种等特点。

LED 点阵按颜色分有单色和双色、全彩三类，可显示红，黄，绿，橙等颜色；LED 点阵按点来排列有 4×4、4×8、5×7、5×8、8×8、16×16、24×24、40×40 等多种；横着有 8 个白色的点，竖着也有 8 个白色的点，就被称为 8x8 的点阵模块，实物图如图 3-49 所示：

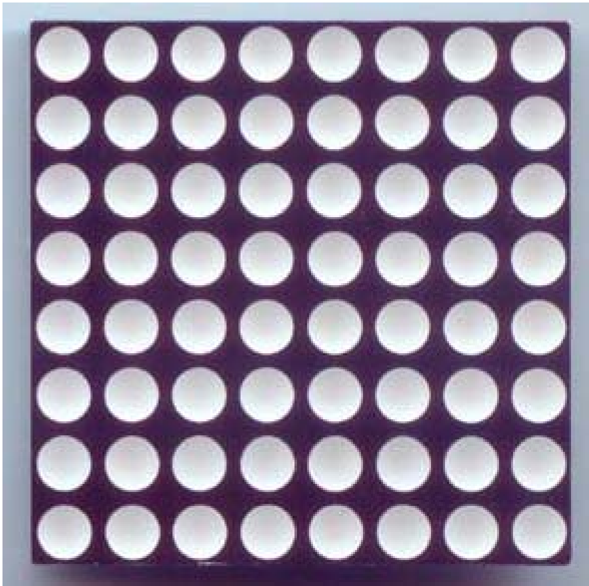


图 3-49 LED 点阵实物图

显示点阵与颜色是不相干的，一个点可能可以显示多个颜色，这个主要看这个点阵的内部结构来决定；例如根据图素的数目可分为双原色、三原色等的，根据图素颜色的不同所显示的文字、图像等内容的颜色也不同，单原色点阵只能显示固定色彩如红、绿、黄等单色，双原色和三原色点阵显示内容的颜色由图素内不同颜色发光二极体点亮组合方式决定，如红绿都亮时可显示黄色，假如按照脉冲方式控制二极体的点亮时间，则可实现 256 或更高级灰

度显示，即可实现真彩色显示。

3.9.2 单色LED点阵的内部结构

下面先分析一个 8x8 的 LED 点阵，可以从图 3-50 看到：

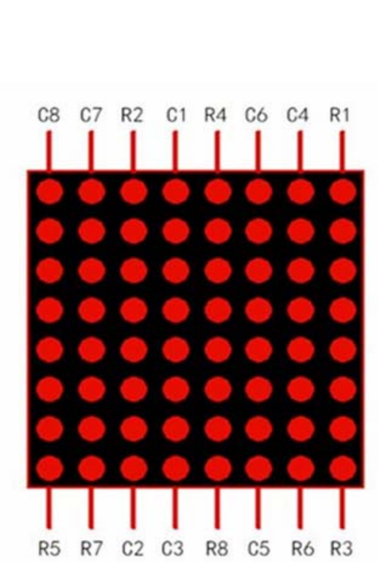


图 3-50 LED 点阵引脚图

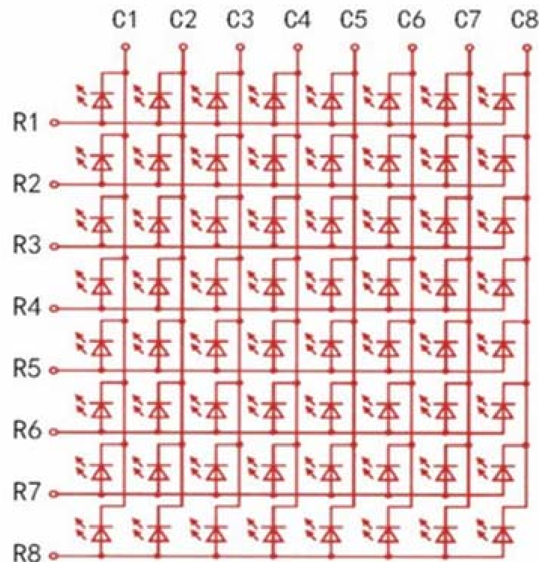


图 3-51 LED 点阵内部结构图

点阵 C1-C8 以及 R1-R8 总共 16 个管脚分布在点阵的两边，一边 8 个管脚；再看图 3-51 可以看到点阵内部的组成原理，点阵内部是由发光二极管排列组成的，它共由 64 个发光二极管组成，且每个发光二极管是放置在行线和列线的交叉点上，当对应的某一行置 1 电平，某一列置 0 电平，则相应的二极管就亮。

如要将第一个点点亮，则 R1 脚接高电平 C1 脚接低电平，则第一个点就亮了；如果要将第一行点亮，则第 R1 脚要接高电平，而（C1、C2、C3、C4、C5、C6、C7、C8）这些引脚接低电平，那么第一行就会点亮；如要将第一列点亮，则第 C1 脚接低电平，而（R1、R2、R3、R4、R5、R6、R7、R8）接高电平，那么第一列就会点亮。

3.9.3 红绿双色LED点阵显示原理

除了上面这种 64 个 LED 组成的单色点阵外，可以看下双色 LED 点阵的结构，如下图 3-52 所示：

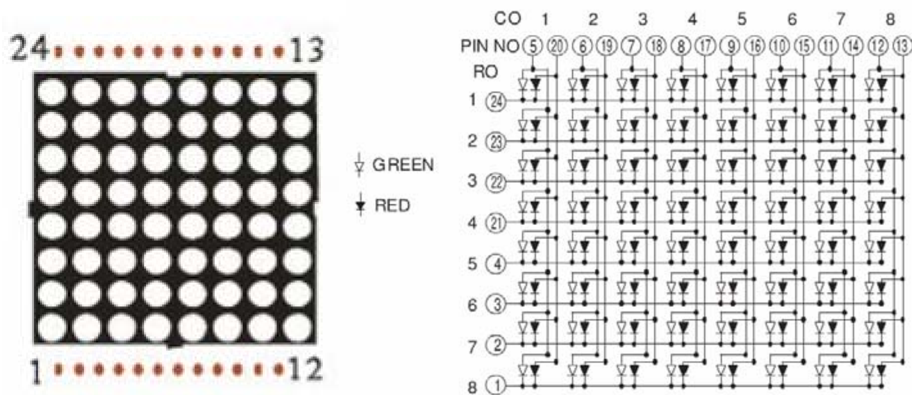


图 3-52 8X8 双色点阵外观图与 8X8 双色点阵功能结构图

单色 8x8 点阵只有 16 个管脚，因为是单色，所以只需要 8 个阳极和 8 个阴极就可以点亮任意一个位置的 LED 二极管；而双色 8x8 点阵模块不同，不仅仅是点亮某个位置的 LED 二极管，还要确定显示的颜色，这样就有了 24 个管脚，8 个红色，8 个绿色，8 个显示管脚来显示，可以从上图看到，双色点阵中的二极管有 128 个，相比单色点阵的 64 个二极管增加了一倍，因为要显示多一种颜色，就要每个点都需要增加一个这样颜色的二极管，可见这个成本是非常高的。

例如

3.9.4 硬件原理图描述

硬件连接如下图 3-53 所示，点阵的绿色显示由 JP25 控制，红色则由 JP26 所控制，另外 J4 连点阵的 12 个管脚，J5 直连点阵的另外 12 个管脚：

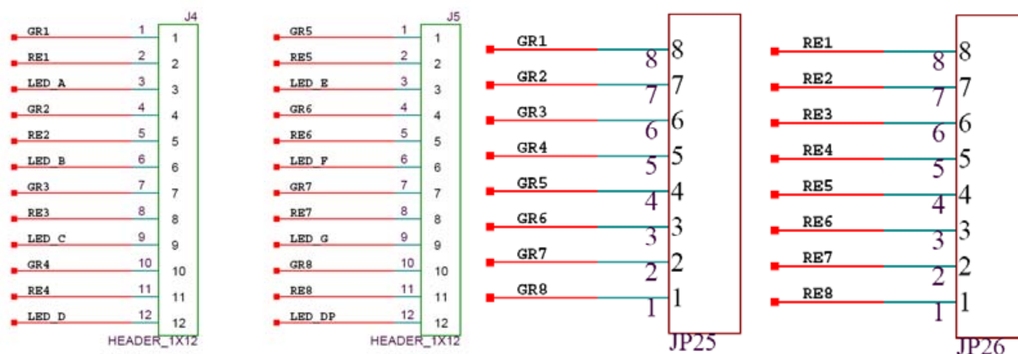


图 3-54 LED 红绿点阵硬件控制连接原理图

而它们的段码则由 JP23 控制锁存器输出，如图 3-54：

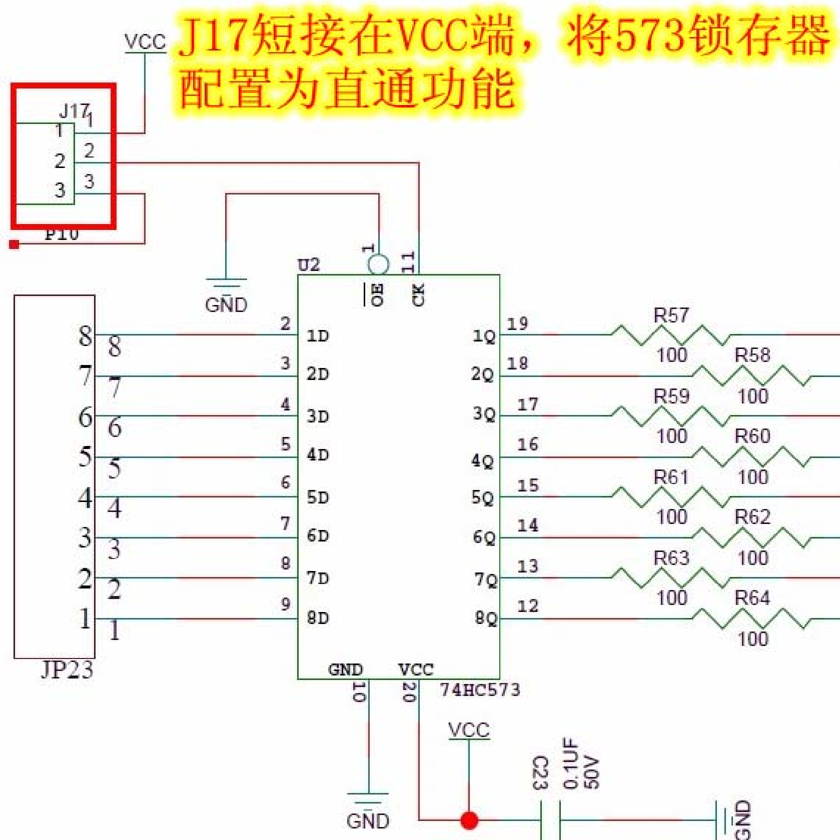


图 3-54 LED 点阵的断码控制

单片机 I/O 口连接 JP23 与 JP25 或者是 JP26，为点阵里面的 LED2 极提供电位，满足 LED 点亮的条件后点亮 LED。JP25 与 JP26 为双色点阵的颜色选择端口。

3.9.5 例程 01 双色点阵 1 种颜色显示 1

代码如下：

```
/******************************************************
* 例程：双色点阵 1 种颜色显示 1
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：双色点阵 1 种红色显示，第一竖条显示全亮
******************************************************/
#include<reg52.h> //包含头文件，一般情况不需要改动，
                    //头文件包含特殊功能寄存器的定义
/*------------------------------------------------
                    主函数
------------------------------------------------*/
void main (void)
{
    while(1)
    {
        P0 = 0xFF;  //横条选择，高电平点亮，即横条 8 个横条灯都亮，0xFF= 1111 1111
```

```
            P2 = 0x7F;   //竖条点阵选择，低电平点亮；即点亮第 1 竖条，0x7F=0111 1111
        }
    }
```

LED 点阵根据硬件原理图连接后的实物图如图 3-55 所示：

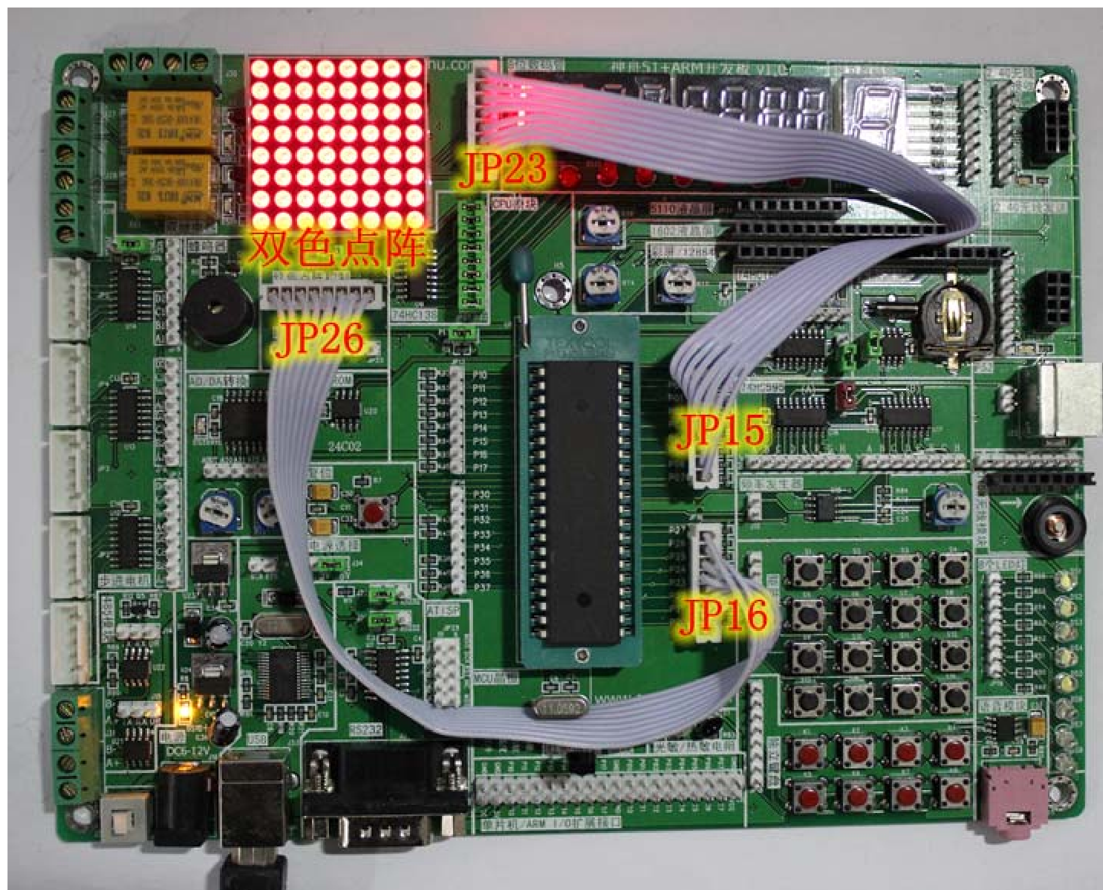


图3-55 LED点阵硬件连接实物图

硬件连接关系如表3-39所示：

表3-39LED点阵硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P0	JP15（A 向左）	直连	JP23(A 向左)	1 根 8 位排线	单片机控制点阵的横
P2	JP16（A 向左）	直连	JP26(A 向上)	1 根 8 位排线	单片机控制红色点阵
J17 短接 VCC 端，再将 J9、J12 的短路帽断开					
实验现象：下载程序后，我们可以看到红色点阵第 1 列点亮					

知识要点：

1. 因为锁存器被配置成了直通模式，直通模式就是锁存器可以被忽略，P0.0 为高锁存器就默认输出高，低就输出低，这就是直通模式，也就是说直接用 P0 接口的 8 个 IO 口就简单灵活控制双色点阵的 8 个管脚管脚。

2. 另外 P2 口控制红色点阵，低电平对应的双色点阵就被点亮。

3. 如果需要点亮绿色，则需要把排线插入到 J25 就可以了，或者用一组新的单片机管脚去控制绿色点阵，就可以灵活的双色点阵显示了。

3.9.6 更多红绿双色点阵例程

更多红绿双色点阵相应的例程可通过北京航空航天大学出版社下载专区下载，如下表3-40：

表 3-40 更多 LED 点阵丰富例程介绍（含详细注释和文档分析）

序号	例程功能
例程 01	双色点阵 1 种颜色显示 1
例程 02	双色点阵 1 种颜色显示 2
例程 03	双色点阵 1 种颜色滚动显示
例程 04	双色点阵 2 种颜色滚动显示
例程 05	双色点阵 2 种颜色滚动显示数字

3.10 串口通讯的收与发

3.10.1 什么是串口通信

串口通信是指外设和计算机间，通过数据信号线 、地线、控制线等，按位进行传输数据的一种通讯方式。这种通信方式使用的数据线少，在远距离通信中可以节约通信成本，但其传输速度比并行传输低。

串口是计算机上一种非常通用的设备通信协议。大多数计算机（不包括笔记本电脑）包含两个基于 RS-232 的串口。串口同时也是仪器仪表设备通用的通信协议(串口通信协议也可以用于获取远程采集设备的数据)。

当年 51 单片机内置串口的时候，被认为是微控制器发展史上的重大事件，因为当时的串口是唯一一个微控制器与 PC 交互的接口。MCU 微控制器经过这么多年的发展，串口仍然是其必不可少的接口之一。

3.10.2串口通信的属性

1. 通信存在的问题

评价一个通信是否优质，主要体现在传输的速度，数据的正确性，功耗是否低，布线成本是否低(例如 1 根线收发都能满足就比 8 根线的并行收发要节约成本)；使用是否普及(就好像大家都学英语，世界很大部分的人都可以独立使用英语吗，会英语的人多，就非常普及，可通信面就非常广；如果你学的鸟语，那就只能跟鸟通信，没有人能听懂)。

2. 串口到底有几个标准？（经常听说有 3 线、5 线串口）

传统的串行接口标准有 22 根线，采用标准 25 芯 D 型插头座（DB25），后来使用简化为 9 芯 D 型插座（DB9），现在应用中 25 芯插头座已很少采用。

像现在所说的几线串口，一般都是指使用了几根线，最初的 RS-232 串口是 25 针的，所

有的针脚定义都有用到，后来变成了 9 针的，所谓全功能串口就是所有的针脚定义都使用上了，例如流量控制，握手信号等都有用到，一般来说国外的产品做产品比较规矩，把所有的串口信号都做上去了。但是国内的技术人员发现，其实 RS-232 串口最主要使用的就是 2，3 线，另外的接口如果不使用的话，也不会出现很大的问题，所以，就在 9 针的基础上做精简，所以就有所谓的 2，3，4，5，6，8 线的串口出来了。.

2 线串口只有 RXD,TXD 两根基本的收发信号线；3 线串口除了 RXD 和 TXD，还有 GND；所谓 4~9 线只是在 TXD 和 RXD 基础上增加了相应的控制信号线，依据实际需要进行设计。

一般来说，使用 5 线的 232 通信，是加了硬件流控的，即 RTS,CTS 信号，主要是为了保证高速通信时的可靠性，如果你的通信速度不是很高，完全可以不用理会。

3. 串口的速度与距离

RS-232（串口的英文代名词）采取不平衡传输方式，即所谓单端通讯。由于其发送电平与接收电平的差仅为 2V 至 3V 左右，所以其共模抑制能力差，再加上双绞线上的分布电容，其传送距离最长为约 15 米，最高速率为 20kb/s。RS-232 是为点对点（即只用一对收、发设备）通讯而设计的，其驱动器负载为 3～7kΩ。所以 RS-232 适合本地设备之间的通信。

4. 从串口通信衍生出 422 与 485 的通信方式

RS-232、RS-422 与 RS-485 都是串行数据接口标准，最初都是由电子工业协会（EIA）制订并发布的，RS-232 在 1962 年发布，命名为 EIA-232-E，作为工业标准，以保证不同厂家产品之间的兼容。

RS-422 由 RS-232 发展而来，它是为弥补 RS-232 之不足而提出的。为改进 RS-232 通信距离短、速率低的缺点，RS-422 定义了一种平衡通信接口，将传输速率提高到 10Mb/s，传输距离延长到 4000 英尺（速率低于 100kb/s 时），并允许在一条平衡总线上连接最多 10 个接收器。RS-422 是一种单机发送、多机接收的单向、平衡传输规范，被命名为 TIA/EIA-422-A 标准。

为扩展应用范围，EIA 又于 1983 年在 RS-422 基础上制定了 RS-485 标准，增加了多点、双向通信能力，即允许多个发送器连接到同一条总线上，同时增加了发送器的驱动能力和冲突保护特性，扩展了总线共模范围，后命名为 TIA/EIA-485-A 标准。

由于 EIA 提出的建议标准都是以“RS”作为前缀，所以在通讯工业领域，仍然习惯将上述标准以 RS 作前缀称谓。

RS-232、RS-422 与 RS-485 标准只对接口的电气特性做出规定，而不涉及接插件、电缆或协议，在此基础上用户可以建立自己的高层通信协议。因此在视频界的应用，许多厂家都建立了一套高层通信协议，或公开或厂家独家使用。如录像机厂家中的 Sony 与松下对录像机的 RS-422 控制协议是有差异的，视频服务器上的控制协议则更多了，如 Louth、Odetis 协议是公开的，而 ProLINK 则是基于 Profile 上的。

5. 串口的通信方式（串口属于串行通信）

（1）并行通信和串行通信

51 单片机与外界通信的基本方式有两种：并行通信和串行通信，并行通信是指利用多条数据传输线将一个数据的各位同时发送或接收。串行通信是指利用一条传输线将数据一位位地顺序发送或接收。

并行通信和串行通信的示意图如下图 3-56：

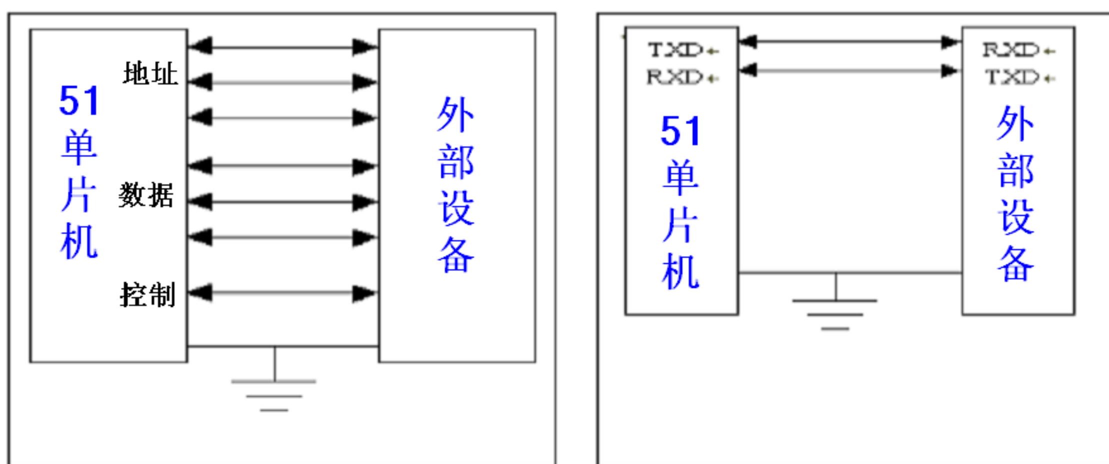


图 3-56 并行通信和串行通信的示意图

在每一条传输线传输速率相同时，并行通信的传输速度比和串行通信快。然而当传输距离变长时，并行通信的缺点就会凸显，首先是相比于串行通信而言信号易受外部干扰，信号线之间的相互干扰也增加，其次是速率提升之后不能保证每根数据线的数据同时到达接收方而产生接收错误，而且距离越长布线成本越高。

所以并行通信目前主要用在短距离通信，比如处理器与外部的 flash 以及外部 RAM 以及芯片内部各个功能模块之间的通信。串行通信以其通信速率快和成本低等优点成为了远距离通信的首选。RS232C 串口，以及差分串行总线像 RS485 串口、USB 接口、CAN 接口、IEEE-1394 接口、以太网接口、SATA 接口和 PCIE 接口等都属于串行通信的范畴。

下图 3-57 左侧为每根数据线的数据同时到达接收方，被正确采样的最理想情况；右侧的图为每根数据线的数据不能同时到达接收方而产生接收错误情形。

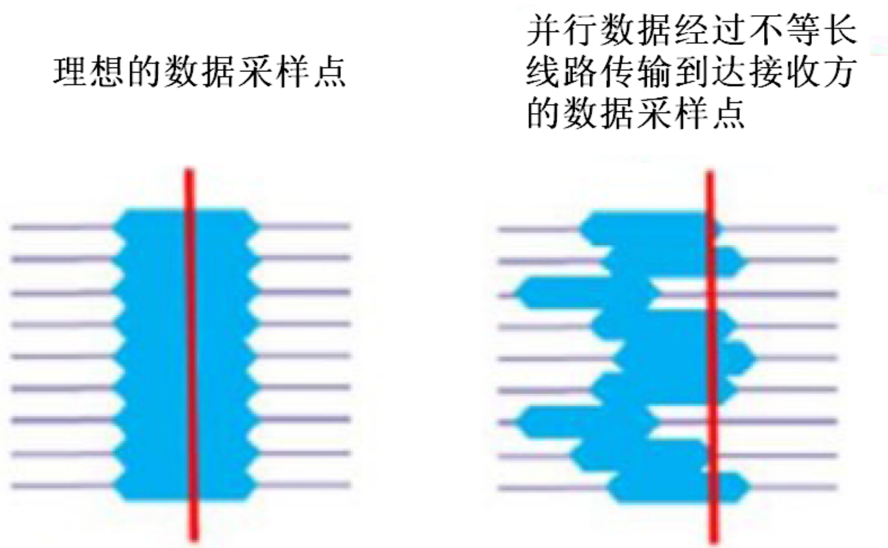


图 3-57 并行通信采样情况

（2）异步通信与同步通信

串行通信又分为两种方式：异步通信与同步通信。

A、 异步通信及其协议

异步通信以一个字符为传输单位，通信中两个字符间的时间间隔不固定可以是任意长的，然而在同一个字符中的两个相邻位代码间的时间间隔是固定的，接收时钟和发送时钟只要相近就可以。 通信双方必须使用约定的相同的一些规则（也叫通信协议）。 常见的传送

一个字符的信息格式规定有起始位、数据位、奇偶校验位、停止位等，其中各位的意义如下图 3-58：

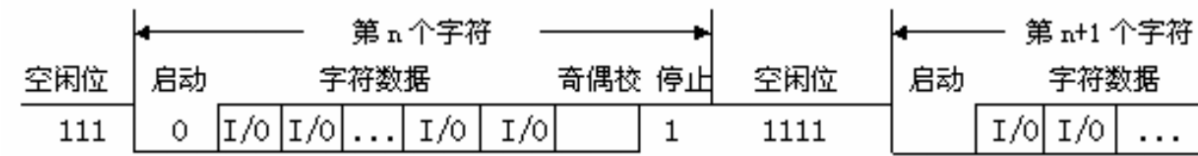


图 3-58 异步通信协议

或者是如下图 3-59

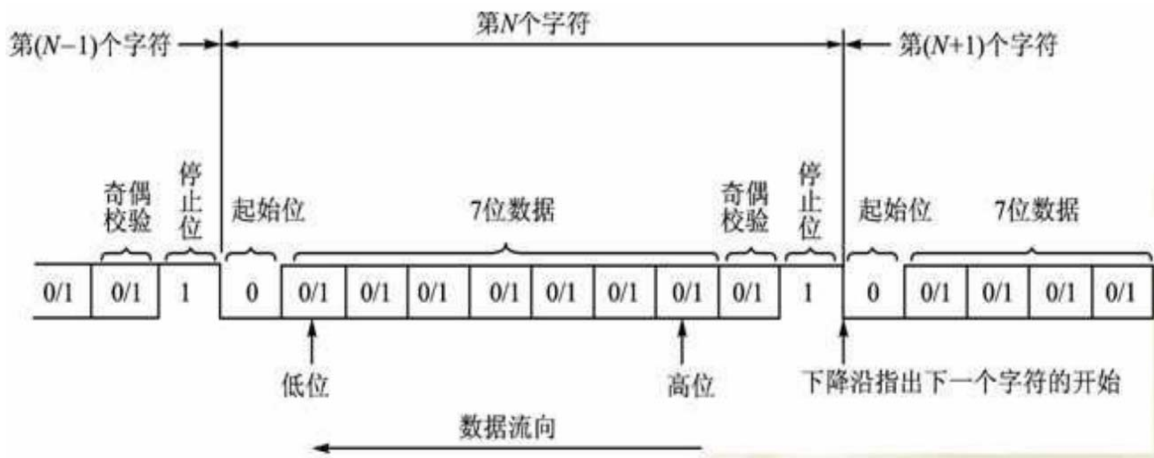


图 3-59 异步通信协议

① 起始位　先发出一个逻辑”0”信号，表示传输字符的开始。

② 数据位　紧接着起始位之后。数据位的个数可以是 5、6、7、8 等，构成一个字符。一般采用扩展的 ASCII 码，范围是 0～255，使用 8 位表示。首先传送最低位。

③ 奇偶校验位（不是必须）　奇偶校验是串口通信中一种简单的检错方式，当然没有校验位也是可以的。数据位加上这一位后，使得“1”的位数应为偶数(偶校验)或奇数(奇校验)，以此来校验数据传送的正确性。例如，如果数据是 01100000，那么对于偶校验，校验位为 0。

④ 停止位　它是一个字符数据的结束标志。可以是 1 位、1.5 位、2 位的高电平。由于数据是在传输线上定时的，并且每一个设备有其自己的时钟，很可能在通信中两台设备间出现了小小的不同步。因此停止位不仅仅是表示传输的结束，并且提供计算机校正时钟同步的机会。适用于停止位的位数越多，不同时钟同步的容忍程度越大，但是数据传输率同时也越慢。

⑤ 空闲位　处于逻辑“1”状态，表示当前线路上没有数据传送。

B、同步通信是指数据传送是以一个帧（数据块或一组字符）为传输单位，每个帧中包含有多个字符。在通信过程中，字符与字符之间、字符内部的位与位之间都同步，每个字符间的时间间隔是相等的，而且每个字符中各相邻位代码间的时间间隔也是固定的。同步通信的数据格式如图 3-60 所示：

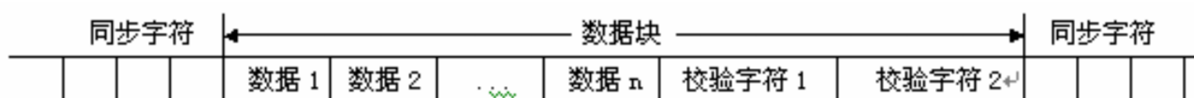


图 3-60 同步通信数据格式

同步通信的特点可以概括为：

① 以数据块为单位传送信息。

② 在一个数据块（信息帧）内，字符与字符间无间隔。

③ 接收时钟与发送进钟严格同步

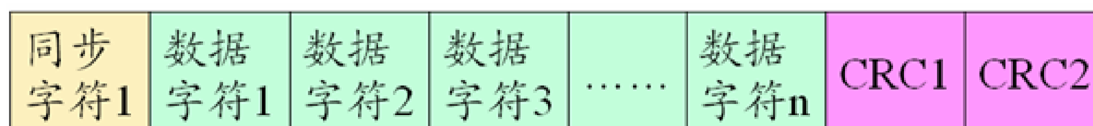
同步串行通信方式中一次连续传输一块数据，开始前使用同步信号作为同步的依据。同步字符的插入可以是单同步字符方式或双同步字符方式，均由同步字符、数据字符和校验字符 CRC 等三部分组成：

同步字符位于帧结构开头，用于确认数据字符的开始。

数据字符在同步字符之后，字符个数不受限制，由所需传输的数据块长度决定；

校验字符有 1～2 个，位于帧结构末尾，用于接收端对接收到的数据字符的正确性的校验。

由于连续传输一个数据块，故收发双方时钟必须相当一致，否则时钟漂移会造成接收方数据辨认错误。这种方式下往往是发送方在发送数据的同时也通过一根专门的时钟信号线同时发送时钟信息，接收方使用发送方的时钟来接由数据。同步串行通信方式传输效率高，但对硬件要求高，电路结构复杂。单同步与双同步字符帧格式如下图 3-61 所示：



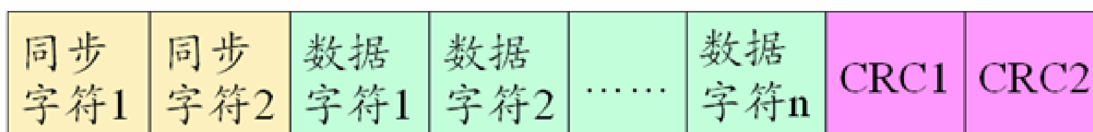


图 3-61 同步传送的数据格式

所有的串行接口电路都是以并行数据形式与 CPU 接口、而以串行数据形式与外部逻辑接口。所以串口对外应该是串行发送的，速度慢，但是比并行传输要稳定很多。

6. 串口是如何解决干扰以及校验的问题

什么是数据校验？通俗的说，就是为保证数据的完整性，用一种指定的算法对原始数据计算出的一个校验值。接收方用同样的算法计算一次校验值，如果和随数据提供的校验值一样，就说明数据是完整的。

为了理解数据校验，什么是最简单的校验呢？最简单的校验就是把原始数据和待比较数据直接进行比较，看是否完全一样这种方法是最安全最准确的。同时这样的比对方式也是效率最低的。只适用于简单的数据量极小的通信。

串口通信使用的是奇偶校验方法，具体实现方法是在数据存储和传输中，字节中额外增

加一个比特位，用来检验错误。校验位可以通过数据位异或计算出来；也就是说单片机串口通讯有一模式就是一次发送 8 位的数据通讯，增加一位第 9 位用于放校验值。

奇偶校验是一种校验代码传输正确性的方法。根据被传输的一组二进制代码的数位中“1”的个数是奇数或偶数来进行校验。采用奇数的称为奇校验，反之，称为偶校验。采用何种校验是事先规定好的。通常专门设置一个奇偶校验位，用它使这组代码中“1”的个数为奇数或偶数。若用奇校验，则当接收端收到这组代码时，校验“1”的个数是否为奇数，从而确定传输代码的正确性。

奇偶校验能够检测出信息传输过程中的部分误码（1 位误码能检出，2 位及 2 位以上误码不能检出），同时，它不能纠错。在发现错误后，只能要求重发。但由于其实现简单，仍得到了广泛使用。

3.10.3 什么是单片机的TTL电平？

单片机是一种数字集成芯片，数字电路中只有两种电平：高电平和低电平；高电平和低电平是通过单片机的管脚进行输入和输出的，我们只要记住一句话，单片机管脚不是输入就是输出，不是高电平就是低电平。

为了让大家在初学的时候对电平特性有一个清晰的认识，我们暂且定义单片机输出与输入为 TTL 电平，其中高电平为+5V，低电平为 0V。计算机的串口出来的为 RS-232C 电平，其中高电平为-5V— -12V，低电平为+5V—+12V。这里要强调的是，RS-232C 电平为负逻辑电平，所以高电平为负的，低电平为正的，大家千万不要认为上面是我写错了，因此当计算机与单片机之间要通信时，需要加电平转换芯片，我们在神舟 51 单片机实验板上所加的电平转换芯片是 MAX3232（在串口 DB9 座附近）。初学者在学习时先掌握上面这点就够了，若有兴趣请大家再看下面的知识点——常用逻辑电平。

知识点：常用逻辑电平

常用的逻辑电平有 TTL、CMOS、LVTTL、ECL、PECL、GTL、RS-232、RS-422、RS-485、LVDS 等。其中 TTL 和 CMOS 的逻辑电平按典型电压可分为四类：5V 系列(5V 的 TTL 和 5V 的 CMOS)、3.3V 系列，2.5V 系列和 1.8V 系列。

5V 的 TTL 和 5V 的 CMOS 是通用的逻辑电平。3.3V 及以下的逻辑电平被称为低电压逻辑电平，常用的为 LVTTL 电平。低电压逻辑电平还有 2.5V 和 1.8V 两种。

那为什么 TTL 电平信号用的最多呢？

原因 1：这是因为大部分数字电路器件都用这个电平标准。就好像我们学英语，国际通用英语这门语言，那大家都用这个语言进行交流和沟通，所以后来的人都要学习英语才能彼此相互能交流。所以使得越来越多的电路器件使用这个电平标准。TTL 电平数据表示通常采用二进制，+5V 等同于逻辑 1，0V 等同于逻辑 0，这被称为 TTL（晶体管一晶体管逻辑电平）信号系统，这是计算机处理器控制的设备内部各部分之间通信的标准技术。TTL 电平信号对于计算机处理器控制的设备内部的数据传输是很理想的，首先计算机处理器控制的设备内部的数据传输对于电源的要求不高，热损耗也较低，另外 TTL 电平信号直接与集成电路连接而不需要价格昂贵的线路驱动器以及接收器电路。

原因 2：TTL 电平的特点适合设备内数据高速的传输。TTL 的通信大多数情况是采用并行数据传输方式，但电平最高为+5V，电压相对比较低，所以传输过程中会有电压损耗和压降，导致 TTL 的传输距离是有限的，一般只适合近距离传输；而且并行数据传输对于超过 10 英尺的距离就可能会有同步偏差，传输距离太远，有可能造成数据不同步；所以 TTL 电平符合近距离（在芯片内部或者计算机内部进行高速数据交互）高速的并行传输，在数字电路要求数据处理速度高的时代来说，选择 TTL 这个标准是正确的，可靠的。

CMOS 电平最高可达 12V，CMOS 电路输出高电平在 3V~12V 之间，而输出低电平接近 0 伏。CMOS 电路中不使用的输入端不能悬空，否则会造成逻辑混乱。另外，CMOS 集成电路因为电源电压可以在较大范围内变化，因而对电源的要求不像 TTL 集成电路那样严格。

TTL 电路和 CMOS 电路的逻辑电平关系如下：

1） CMOS 是场效应管构成，TTL 为双极晶体管构成；因为 TTL 和 COMS 的高低电平的值不一样,所以互相连接时需 要电平的转换。

2） TTL 电路是电流控制器件，而 coms 电路是电压控制器件。

3） TTL 电路的速度快，传输延迟时间短(5-10ns)，但是功耗大；COMS 电路的速度慢，传输延迟时间长(25-50ns),但功耗低，COMS 电路本身的功耗与输入信号的脉冲频率有关，频率越高，芯片集越热，这是正常现象。

4） CMOS 集成电路电源电压可以在较大范围内变化，因而对电源的要求不像 TTL 集成电路那样严格。所以，用 TTL 电平在条件允许下他们就可以兼容。要注意到他们的驱动能力是不一样的,CMOS 的驱动能力会大一些,有时候 TTL 的低电平触发不了 CMOS 电路,有时 CMOS 的高电平会损坏 TTL 电路，在兼容性上需注意。

5） CMOS 的高低电平之间相差比较大、抗干扰性强，TTL 则相差小，抗干扰能力差。

6） CMOS 的工作频率较 TTL 略低。

TTL 电平临界值：

1）TTL 输出电压：逻辑电平 1 = 2.4V，逻辑电平 0 = 0.4V

2）TTL 输入电压：逻辑电平 1 = 2.0V，逻辑电平 0 = 0.8V

CMOS 电平临界值（设电源电压为+5V）

1） CMOS 输出电压：逻辑电平 1 = 4.99V，逻辑电平 0 = 0.01V

2） CMOS 输入电压：逻辑电平 1 = 3.5V，逻辑电平 0 = 1.5V

常用逻辑芯片的特点如下：

74LS 系列属于 TTL 器件，其中输入和输出电平模式都是 TTL 电平

74HC 系列属于 CMOS 器件，其中输入和输出电平模式都是 CMOS 电平

CD4000 系列属于 CMOS 器件，其中输入和输出电平模式都是 CMOS 电平

74HCT 系列属于 CMOS 器件，其中输入电平模式是 TTL 电平，输出电平模式都是 CMOS 电平

通常情况下，单片机、ARM、DSP、FPGA 等各个器件之间引脚能否直接相连要参考以下方法进行判断：一般来说，同电压的是可以相连的，不过最好还是好好查看芯片技术手册上的 VIL（逻辑电平 0 的输入电压）、VIH（逻辑电平 1 的输入电压）、VOL（逻辑电平 0 的输出电压）、VOH（逻辑电平 1 的输出电压）的值，看是否能够匹配。有些情况在一般应用中没有问题，虽然参数上有点不够匹配，但还是在管脚的最大和最小容忍值范围之内，不过有可能在某些情况下可能就不够稳定，所以我们在设计电路的时候要尽量保持匹配，这样是最佳的设计。

3.10.4 关于NPN和PNP的三极管基础知识？

对三极管放大作用的理解，切记一点：能量不会无缘无故的产生，所以，三极管一定不会产生能量，但三极管厉害的地方在于：它可以通过小电流控制大电流。放大的原理就在于：通过小的交流输入，控制大的静态直流。假设三极管是个大坝，这个大坝奇怪的地方是，有两个阀门，一个大阀门，一个小阀门。小阀门可以用人力打开，大阀门很重，人力是打不开的，只能通过小阀门的水力打开。所以，平常的工作流程便是，每当放水的时候，人们就打

开小阀门，很小的水流涓涓流出，这涓涓细流冲击大阀门的开关，大阀门随之打开，汹涌的江水滔滔流下。如果不停地改变小阀门开启的大小，那么大阀门也相应地不停改变，假若能严格地按比例改变，那么，完美的控制就完成了。

在这里，基极B->发射极E就是小水流，集电极C->发射极E就是大水流，如图3-62所示。当然，如果把水流比为电流的话，会更确切，因为三极管毕竟是一个电流控制元件。



图 3-62　三极管介绍

如果某一天，天气很旱，江水没有了，也就是大的水流那边是空的。管理员这时候打开了小阀门，尽管小阀门还是一如既往地冲击大阀门，并使之开启，但因为没有水流的存在，所以，并没有水流出来。这就是三极管中的截止区。

饱和区是一样的，因为此时江水达到了很大很大的程度，管理员开的阀门大小已经没用了。如果不开阀门江水就自己冲开了，这就是二极管的击穿。

在模拟电路中，一般阀门是半开的，通过控制其开启大小来决定输出水流的大小。没有信号的时候，水流也会流，所以，不工作的时候，也会有功耗。

而在数字电路中，阀门则处于开或是关两个状态。当不工作的时候，阀门是完全关闭的，没有功耗。

那么NPN与PNP的三极管到底有些什么区别呢？二者之间的区别如图3-63所示"

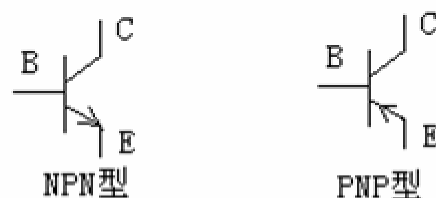


图3-63 三极管不同管型的区别

NPN和PNP主要就是电流方向和电压正负不同，说得“专业”一点，就是“极性”问题。

NPN 是用 B→E 的电流（小水流）控制 C→E 的电流（大水流），E极电位最低，且正常放大时通常C极电位最高，即 VC>VB>VE。

PNP 是用 E→B 的电流（小水流）控制 E→C 的电流（大水流），E极电位最高，且正常放大时通常C极电位最低，即VC<VB<VE。

半导体三极管也称为晶体三极管，可以说它是电子电路中最重要的器件。它最主要的功能是电流放大和开关作用。接下来的一些使用中会用到。

3.10.5 RS-232电平与TTL电平的转换

关于RS-232电平与TTL电平的特性在前面已经讲过，本节主要讲解使用较多的计算机RS-232电平与单片机TTL电平之间的转换方式。MAX232等芯片可实现RS-232电平到

TTL电平的转换，但是现在用的较多还有MAX202，HIN232等芯片，它们同时集成了RS-232电平和TTL电平之间的互转。为丰富大家的知识，下面首先讲解在没有MAX3232这种现成电平转换芯片时，如何用二极管、三极管、电阻、电容等分立元件搭建一个简单的RS-232电平与TTL电平之间的转换电路。

1.用单独的电容电阻三极管实现RS-232电平与TTL电平转换电路

集成芯片内部都是由最基本电子元件组成，如电阻、电容、二极管、三极管等元件，为了方便用户使用，制造商把这些具有一定功能的分立元件封装到一个芯片内，这样就制成了我们使用的各种芯片。学会本电路后，我们也就基本搞清了 MAX232 芯片内部的大致结构。

MAX232是把TTL电平从0V~5V转换到3V～15V或-3V～-15V之间。如下图3-64所示：

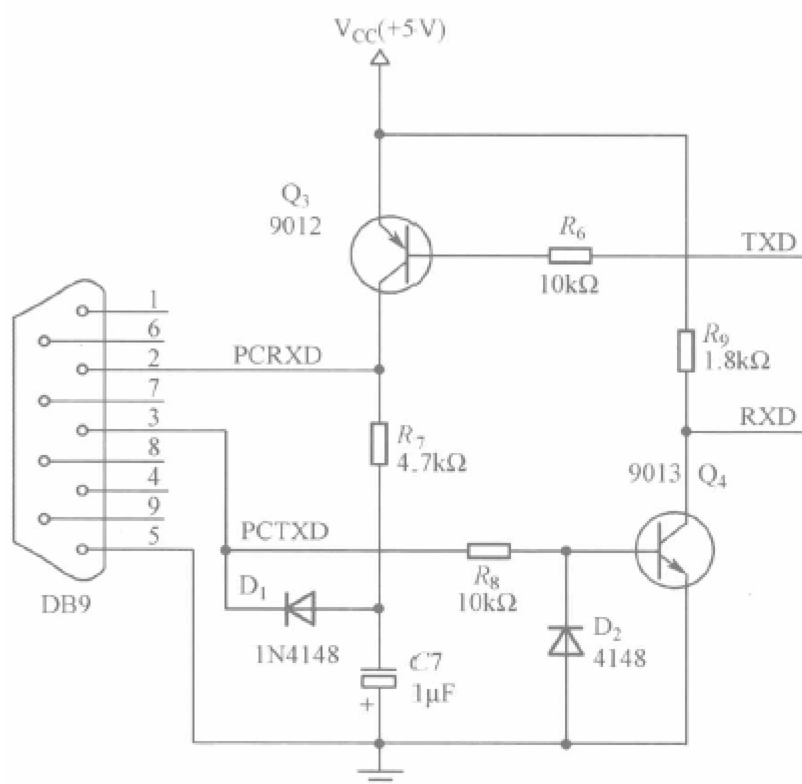


图 3-64 MAX232 电路原理图

(1) 若发送低电平0，首先TXD（TTL低电平）发送数据时，TXD上是低电平，这时Q3导通（具体请看上节三极管的描述），PCRXD由空闲时的低电平变高电平，满足条件。

(2) 发送高电平1时，TXD为高电平，Q3截止，由于PCRXD内部高阻，而PCTXD平时是-3～-15V（RS-232的高电平就是负的电压，这点是要注意的，高电平并不是正电压），通过D1和R7将其拉低PCRXD至-3～-15V，此时计算机接收到的就是1。

下面再反过来,PC发送信号,由单片机来接收信号。当PCTXD为低电平-3～-15V时，Q4截止，单片机端的RXD被R9拉到5V高电平；当PCTXD变高时，Q4导通，RXD被Q4拉到低电平，这样便实现的双向转换，这是一个很好的电路，值得大家学习。

2. MAX232芯片实现RS-232电平与TTL电平转换

MAX232 芯片是 MAXIM 公司生产的、包含两路接收器和驱动器的 IC 芯片，它的

内部有一个电源电压变换器，可以把输入的+5V 电源电压变换成为 RS-232 输出电平所需的+10V 电压。所以，采用此芯片接口的串行通信系统只需单一的+5V 电源就可以了。对于没有+12V 电源的场合，其适应性更强，加之其价格适中，硬件接口简单，所以被广泛采用。

MAX232芯片实物和其引脚结构和外围连接如下图3-65所示：



图 3-65 MAX232 芯片实物图

而它的引脚内部结构图如图 3-66 所示：

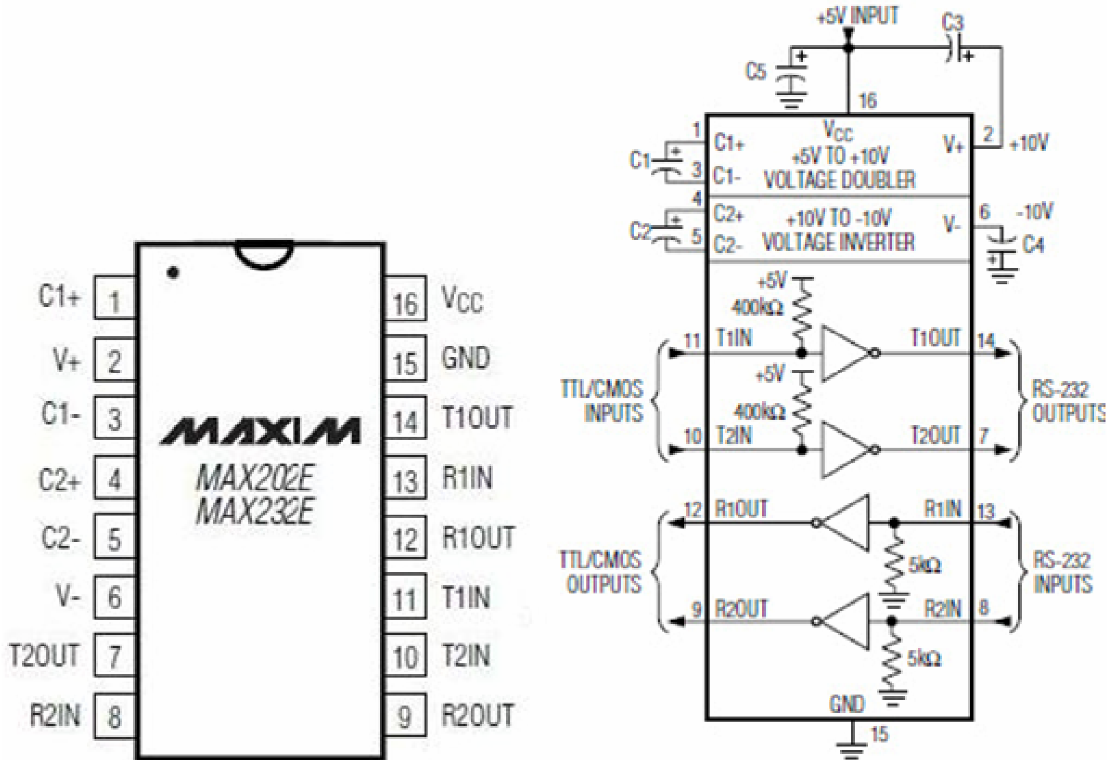


图 3-66　MAX233 脚内部结构图

在上图中上半部分电容 Cl，C2，C3，C4 及 V+，V-是电源变换电路部分。在实际应用中，器件对电源噪声很敏感，因此 VCC 必须要对地加去耦电容 C5，其值为 O.luF。按芯片手册中介绍，电容 Cl，C2，C3，C4 应取 1.OuF/16V 的电解电容，经大量实验及实际应用，这 4 个电容都可以选用 O.luF 的非极性瓷片电容代替 l.OuF/16V 的电解电容，在具体设计电路时，这 4 个电容要尽量靠近 MAX232 芯片，以提高抗干扰能力。

图下半部分为发送和接收部分。实际应用中,T2IN，R2OUT 可直接连接 TTL/CMOS 电平的 51 单片机主芯片的串口发送端 TXD_RS232 和 RXD_RS232；T2out，R2IN 可直接连接 PC 机的 RS-232 串口的接收端 RXD 和发送端 TXD。

MAX3232与51单片机连接的是TTL电平（0V-5V），而与PC连接的是RS232电平

（-15V--+15V），所有这颗芯片完成了一个电平的转换功能，具体的原理图说明如下图3-67所示：

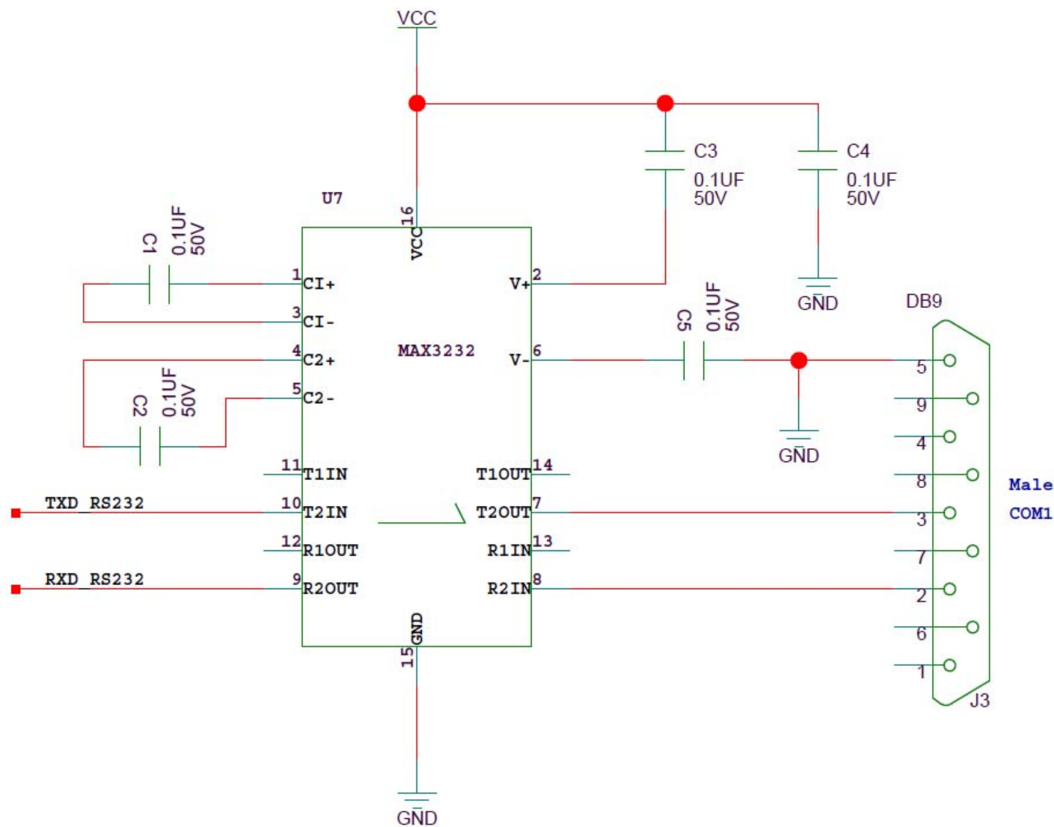


图 3-67 MAX232 硬件原理图

神舟 51 开发板上对应于MAX232 芯片的是：RS-232C标准的DB9 串口公头插针。其线序与PC电脑上的DB9 公头插针相同，都为 2 脚输入到开发板，3 脚输出，如下图 3-68 所示。所以只要一根母到母交叉串口线就可以很方便的将其与PC电脑的串口连接起来，或者将两个开发板上的串口对连起来。

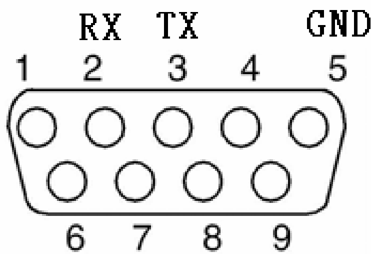


图 3-68 DB9 串口公头座子

DB9 串口公头插针线序如下表 3-41 所示，其信号定义如下表所示，总共 9 根信号线：

表 3-41 DB9 公头插针引脚信号定义

DB9管脚	功能描述	DB9管脚	功能描述
第1脚	NC	第6脚	NC

第2脚	UART_RXD 开发板接收	第7脚	NC
第3脚	UART_TXD 开发板发送	第8脚	NC
第4脚	NC	第9脚	NC
第5脚	GND 开发板电源地		

3.10.6 神舟51+ARM独特的USB转串口的TTL电平模块设计

RS232 接口作为标准外设广泛应用于单片机和嵌入式系统，通用串行总线 USB(Universal Serial Bus)通信技术以其易插拔、速度快、即插即用和独立供电等特点，已得到更广泛的应用。

STM32 神舟 51+ARM 采用了一种基于 PL2303 的 RS232 与 USB 转换的设计方案，作为开发板与电脑串口之间的交互接口。PL2303 是高集成度的通用串行总线(USB)与串口的接口转换器，可方便将现有基于 RS232 接口的设备转换为 USB 接口。

PL2303 是 Prolific 公司生产的一种高度集成的 RS232-USB 接口转换器，可提供一个 RS232 全双工异步串行通信装置与 USB 功能接口便利联接的解决方案。该器件内置 USB 功能控制器、USB 收发器、振荡器和带有全部调制解调器控制信号的 UART，只需外接几只电容就可实现 USB 信号与 RS232 信号的转换，能够方便嵌入到手持设备。该器件作为 USB／RS232 双向转换器，一方面从主机接收 USB 数据并将其转换为 RS232 信息流格式发送给外设；另一方面从 RS232 外设接收数据转换为 USB 数据格式传送回主机。这些工作全部由器件自动完成，开发者无需考虑固件设计。

市场上主要的 USB 转串口芯片有 FT232、PL2303、CH340 三种，三个常用的芯片稳定程度和价格是一致的，FT232>CH340>PL2303，PL2303 用的最多，因为最便宜，国内很多开发板板子上，包括 USB 转串口线用的都是这种芯片，几元钱一片，电路也简单，做简单的串口应用可以，但是做嵌入式开发如使用超级终端波特率在 115200 时就有可能出现延迟等现象。CH340 是南京沁恒的芯片，做的还不错，对于普通应用完全能够满足。最好的是 FT232 稳定、可靠，在很多 USB 转串口的下载线、编程器中使用的都是这一种，神舟开发板上目前使用的是 PL2303HX 芯片。

USB 转串口芯片转出来串口电平就是 TTL 电平，高电平一般是 3.3V，如果转出来的电平再经过 MAX232 或 MAX485 芯片再转一下就会输出 RS232 电平或者 485 电平。其实市面上的 USB 转串口线一般都是这样接的。

3.10.7 串口波特率的理解

在信息传输通道中，携带数据信息的信号单元叫码元，每秒钟通过信道传输的码元数称为码元传输速率，简称波特率。波特率是指数据信号对载波的调制速率，它用单位时间内载波调制状态改变的次数来表示(也就是每秒调制了符号数)，其单位是波特（Baud, symbol/s）。波特率是传输通道频宽的指标。

它是对信号传输速率的一种度量。但是波特率有时候会同比特率混淆，实际上后者是对信息传输速率（传信率）的度量。当 1 波特等于 1 比特的时候，波特率与比特率才相等；但是如果 1 波特等于 8 比特的时候，那么每秒钟发送的比特率是波特率的 9 倍，波特率可以被理解为单位时间内传输码元符号的个数（传符号率），通过不同的调制方法可以在一个码元上负载多个比特信息。

所以，如果用公式表示，比特率在数值上和波特率有这样的关系：

波特率与比特率的关系为：比特率=波特率 X 单个调制状态对应的二进制位数。

单片机或计算机在串口通信时的速率用波特率表示，它定义为每秒传输二进制代码的位数，即 1 波特=1 位 / 秒，单位是 bps（位 / 秒）。如每秒钟传送 240 个字符，而每个字符格式包含 10 位（1 个起始位、1 个停止位、8 个数据位），这时的波特率为 10 位×240 个 / 秒= 2400 bps。

串行接口或终端直接传送串行信息位流的最大距离与传输速率及传输线的电气特性也有关。当传输线使用每0.3m（约1英尺）有50pF电容的非平衡屏蔽双绞线时，传输距离随传输速率的增加而减小。当比特率超过1000 bps时，最大传输距离迅速下降，如9600 bps时最大距离下降到只有76m（约250英尺）。因此我们在做串口通信实验选择较高速率传输数据时，尽量缩短数据线的长度，为了能使数据安全传输，即使是在较低传输速率下也不要使用太长的数据线。

3.10.8 51 单片机内部的UART串口简介

51 内部有一个可编程全双工串行接口,具有 UART(通用异步接收和发送器)的全部功能，该串行口有 4 种工作方式，以供不同场合使用。波特率可由软件设置，通过对串口编程，可以实现串半并转换，双机通信及多机通信。

串行口数据缓冲器 SBUF，51 单片机上有两个物理上独立的接收、发送缓冲器 SBUF(属于特殊功能寄存器)：一个用作发送；一个用作接收。发送缓冲器只能写入不能读出，写入的数据存储在 SBUF 发送缓冲器，用于串行发送；接收缓冲器只能读出不能写入。两者共用一个字节地址(99H)。串行口结构如下图 3-69 所示。

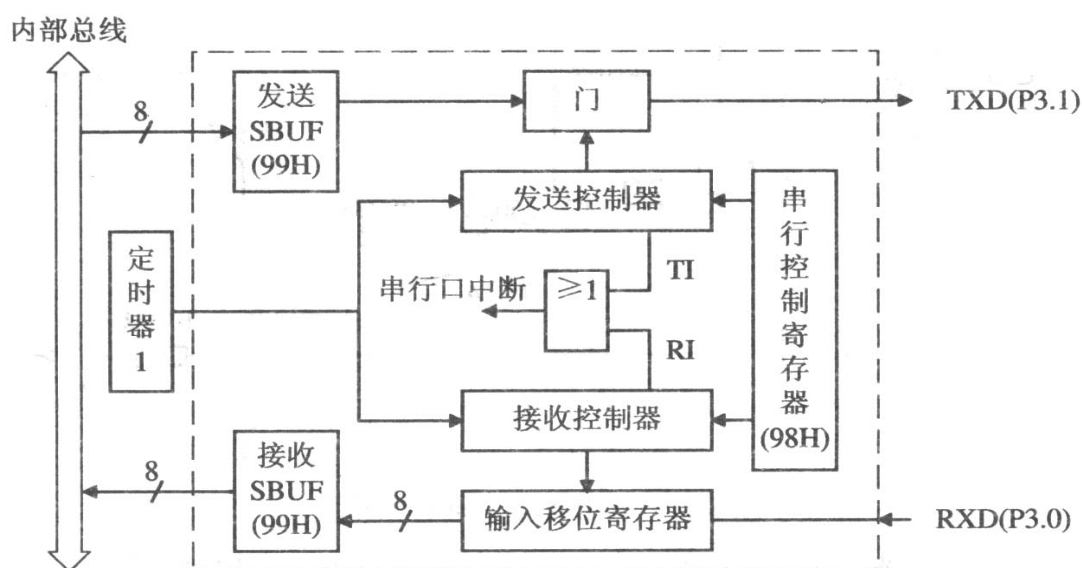


图 3-69 串行口结构图

通过对 SBUF 的读、写指令来区别是对接收缓冲器还是发送缓冲器进行操作。接收或发送数据，是通过串行口对外的两条独立收发信号线 RXD(P3.0)、TXD(P3.1)来实现的。因此可以同时发送、接收数据，实现全双工。

在发送时，CPU 由一条写发送缓冲器的指令把数据(字符)写入串行口的发送缓冲器 SBUF(发)中，然后从 TXD 端一位位地向外发送。

与此同时，接收端 RXD 也可一位位的接收数据，直到收到一个完整的字符数据后通知 CPU，再用一条指令把接收缓冲器 SBUF(收)的内容读入累加器。

51 有一个可通过软件控制的内置全双工串行通讯接口。它由可位寻址的寄存器 SCON 来进行设置。

SCON 的结构如下表 3-42：

表 3-42 SCON 寄存器结构

SCON	D7	D6	D5	D4	D3	D2	D1	D0
位地址	9FH	9EH	9DH	9CH	9BH	9AH	99H	98H
位符号	SM0	SM1	SM2	REN	TB8	RB8	TI	RI

各位功能如表 3-43 所示。

表 3-43 SCON 寄存器各位功能

位符号	功　能　说　明
SM0，SM1	SM0，SM1——串行口工作方式选择位： SM0 SM1 工作方式 功能 0 0 0 8 位数据传送，波特率固定，为 f_{osc}/12。（f_{osc} 为主机频率） 0 1 1 10 位数据传送，波特率可变。 1 0 2 11 位数据传送，波特率固定，为 f_{osc}/64 或 f_{osc}/32。 1 1 3 11 位数据传送，波特率可变。
SM2	SM2——多机通信控制位： 串行口以方式 2 或方式 3 接收时，如 SM2＝1，则只有当接收到的第 9 位数据（RB8）为 1，才将接收到的前 8 位数据送入接收 SBUF，并使 RI 位置 1，产生中断请求信号；否则将接收到的前 8 位数据丢弃。而当 SM2＝0 时，则不论第九位数据为 0 还是为 1，都将前 8 位数据装入接收 SBUF 中，并产生中断请求信号。对方式 0，SM2 必须为 0， 对方式 1，当 SM2=1，当接收到有效停止位后使 RI 位置 1。
REN	REN——允许接收位，用于对串行数据的接收进行控制： REN＝0， 禁止接收；REN＝1，允许接收。该位由软件置 1 或清零。
TB8	TB8——发送数据第 9 位： 在方式 2 和方式 3 时，TB8 是要发送的第 9 位数据。
RB8	RB8——接收数据第 9 位： 在方式 2 和方式 3 中，RB8 位存放接收到的第 9 位数据
TI	TI——发送中断标志： 当方式 0 时，发送完第 8 位数据后，该位由硬件置位。在其它方式下，于发送停止位之前由硬件置位。因此 TI＝1，表示帧发送结束。其状态既可供软件查询使用，也可请求中断。 TI 位由软件清 0。
RI	RI——接收中断标志： 当方式 0 时，接收完第 8 位数据后，该位由硬件置 1。在其它方式下，当接收到停止位时，该位由硬件置 1。因此 RI＝1，表示帧接收结束。其状态既可供软件查询使用，也可以请求中断。RI 位由软件清 0。

51 单片机支持 10 位和 11 位数据模式。11 数据模式用来进行多机通讯并支持高速 8 位移位寄存器模式。模式 1 和模式 3 中波特率可变个数据位，1 个停止位，这种方式可和包括 PC 机在内的很多器件进行通讯，这种方式中波特率是可调的，而用来产生波特率的定时器的中断应该被禁止，PCON 的 SMOD 位为 1 时可使波特率翻倍。

本章节实验串口工作在方式 1，以下介绍方式 1 的特性：

工作方式 1 为波特率可变的 8 位通用异步通信接口。当 SCON 中的 SM0、SMl 两位为 01 时，串行口以方式 1 工作，此时串行口为 8 位异步通信接口，波特率可变。一帧格式为 10 位：1 位起始位，8 位数据位(低位在前)和一位停止位。TXD 为发送端，RXD 为接收端。如图 3-70 所示：

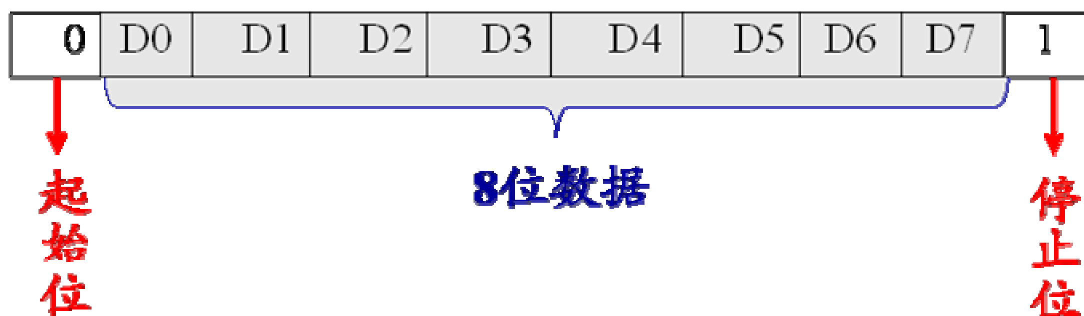


图 3-70 串行口工作方式 1

（1）数据发送

发送时，数据从 TXD 端输出。当执行 SBUF = ‘c’指令时，数据被写入发送缓冲器 SBUF，并启动发送器发送。当发送完一帧数据后，置中断标志 TI 为 1。

（2）数据接收

当串行口置为方式 l，且 REN=I 时，串行口处于方式 1 的接收状态。它以所选波特率的 16 倍的速率对 RXD 引脚状态采样。当采样到由 1 到 0 跳变时，确认是起始位“0”，启动接收器开始接收一帧数据。

当 RI=0 且接收到停止位为 1（或 SM2=0）时，将停止位送入 RB8，8 位数据送入接收缓冲器 SBUF，同时置中断标志 RI=1。

所以，方式 1 接收时，应先用软件清除 RI 或 SM2 标志。若上述两个条件不满足，信息将丢失。这时将重新检测 RXD 上 1 到 0 的负跳变，以接收下一帧数据。中断标志 RI 必须由用户在中断服务程序中清零。

（3）收发波特率

在方式 1 下，波特率由定时器 Tl 的溢出率和 SMOD 共同决定，因而波特率也是可变的。相应公式为：

波特率=2SMOD /32× n(定时器 T1 的溢出率)

溢出率为溢出周期的倒数，所以波特率为：

$$\text{波特率} = \frac{2^{SMOD}}{32} \cdot \frac{f_{OSC}}{12(2k - X)}$$

定时器 T1 的溢周期为：fOSC /12×1/(2n-X)，其中 fOSC 为晶振频率，n 为定时器 T1 的位数，它和定时器 T1 的设定方式有关。定时器 T1 通常采用方式 2，则 n=8。

定时器 T1 方式 2 时 TH1 和 TLl 分别设定为两个 8 位重装计数器(当 TLl 从全“1”变为全“0”时，TH1 重装 TL1)。这种方式，不仅可使操作方便，也可避免因重装初值（时间常数初值）而带来的定时误差。方式 1 下所选波特率常常需要通过计算来确定初值。

（4）串行口中断

串行口中断的中断号为 4，如下表 3-44 所示：

表 3-44 中断源对应的中断号介绍

IE 寄存器中使能位和 C 中的中断号	中断源
0	外部中断 0
1	定时器 0 溢出
2	外部中断 1
3	定时器溢出 1
4	串行口中断
5	定时器 2 溢出

3.10.9 单片机串口硬件连接原理

RS232（计算机的）是用正负电压来表示逻辑状态，与 TTL（51 单片机）高低电平表示逻辑状态的，规定不同。因此，51 单片机能够同计算机接口相连时，必须在 RS232 与 TTL 之间进行电平和逻辑关系的转换。神舟 51+ARM 开发板用 MAX3232 集成芯片实现这种转换。

同样的道理计算机和单片机通过USB接口通信的话，数据也需要转换。为此神舟 51+ARM开发板提供了PL2303 集成芯片。

神舟 51 开发板上对应于PL2303 芯片的是：将单片机的TTL数据通过芯片PL2303 转换的USB接口。它可以很方便的连接到PC电脑的USB接口，还可以通过PC电脑的USB接口为神舟 51 开发板提供 5V电源。

USB 接口外形图如图 3-71 所示：

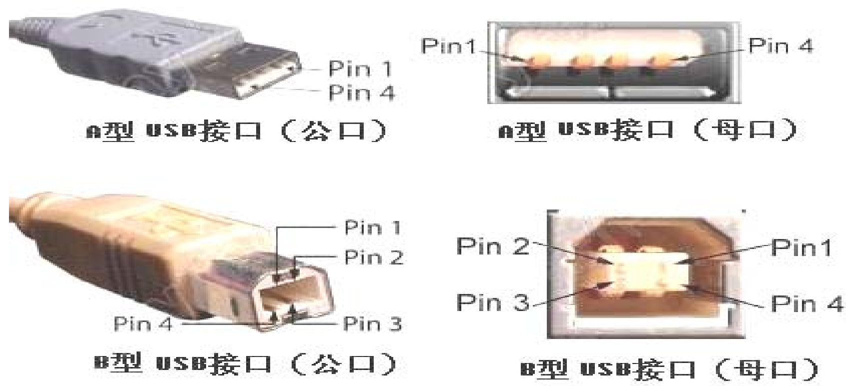


图 3-71 USB 接口外形图

USB 引脚定义如表 3-45 所示：

表 3-45 USB 引脚定义

引脚	定义	符号
1	5 伏直流电压	VCC
2	数据线	D-
3	数据线	D+
4	信号地	GND

硬件原理图如图 3-72 所示：

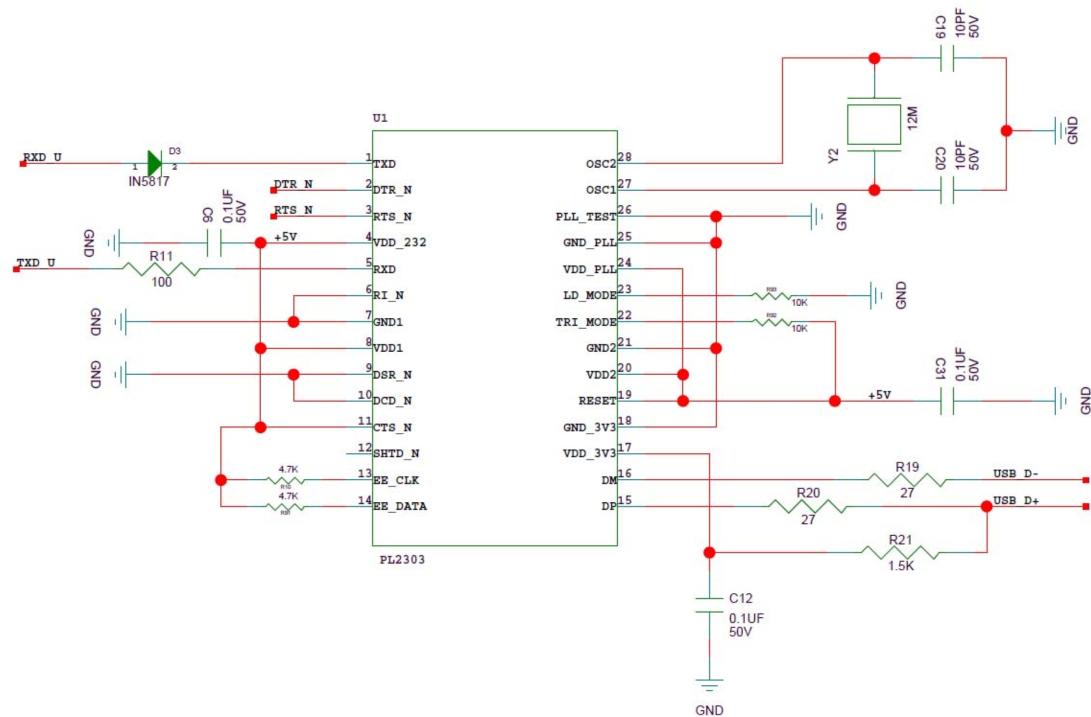


图 3-72 单片机串口硬件原理图

神舟 51 开发板上的单片机串口有两种形式的对外接口，通过跳帽来选择使用其中的一种。如下图 3-73 设置即可

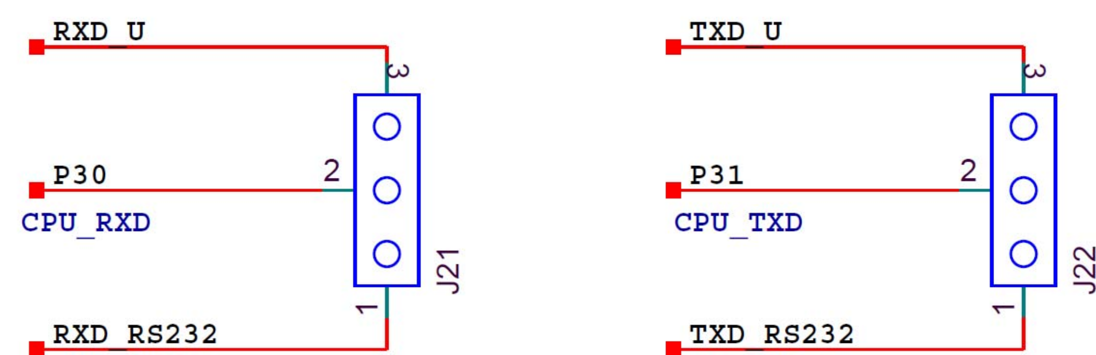


图 3-73 单片机串口形式选择

将神舟 51 开发板上的串口用以上任意一种方式与PC电脑连接以后，在PC上打开超级终

端。打开方式为：Window下点击左下角的“开始”-->程序-->附件-->通讯-->超级终端，如下图 3-74 所示：

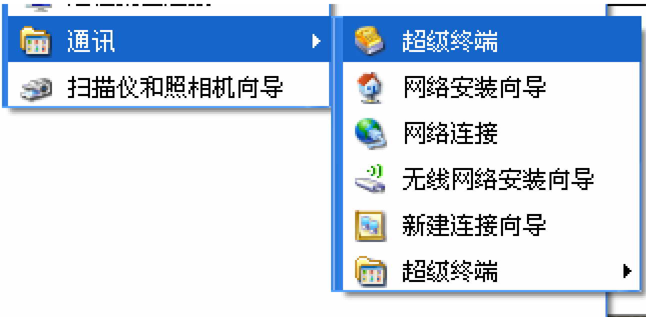


图 3-74 打开串口通信工具——超级终端（其他串口通信工具也可以）

按如下图 3-75 方式设置参数：

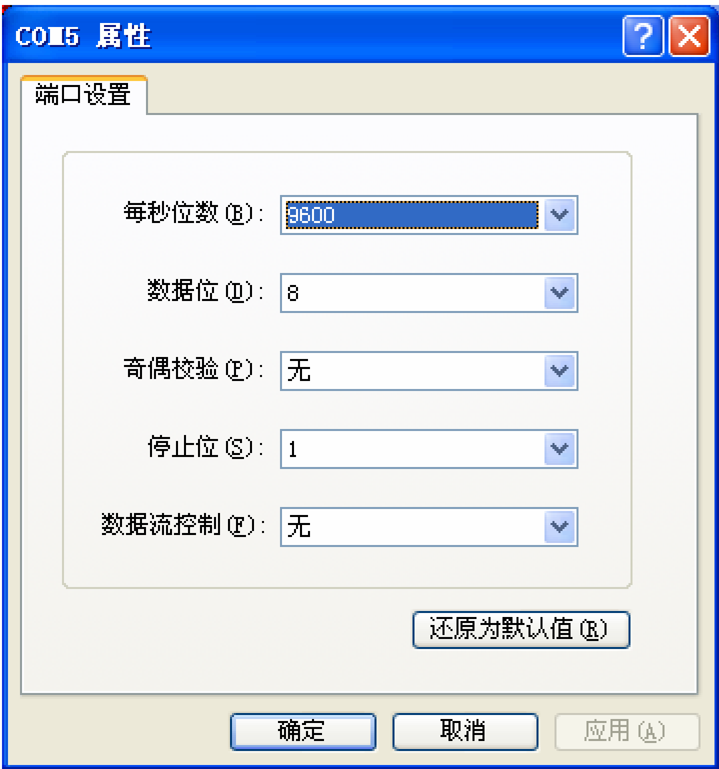


图 3-75 串口通信工具的设置

3.10.10 例程 01 DB9 串口输出一个字符

代码如下：

```
/***********************************************************************
* 例程：DB9串口输出一个字符
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：通过单片机的串口向PC发送一个字符'c'，在PC的超级终端中查看串口消息
* 现象：通过本例程了解单片机串口的基本原理，理解并掌握单片机的串口通信知识
```

```
*           连接好串口或者usb转串口至电脑，下载该程序，打开电脑的超级终端工具
*           将波特率设置为9600，无奇偶校验，设置正确后可以看到接收到的测试字符
* 注意：晶振11.0592MHz
*           波特率为9600
*           使用DB9串口，检查J21和J22的跳帽位置
*           使用跳帽短接 J21.RXD   与   J21.RS232
*           使用跳帽短接 J22.TXD   与   J22.RS232。
***************************************************************************/
/* 包含头文件，一般情况不需要改动，头文件包含特殊功能寄存器的定义 */
#include <reg52.h>
#include "delay.h"
/*------------------------------------------------------------
                    主函数
------------------------------------------------------------*/
void main (void)
{
    //串口初始化
    SCON   = 0x50;          // SCON:  模式 1, 8-bit UART,  使能接收
    TMOD |= 0x20;           // TMOD: timer 1, mode 2, 8-bit  重装
    TH1    = 0xFD;          // TH1:   重装值  9600  波特率  晶振  11.0592MHz
    TR1    = 1;             // TR1:   timer 1  打开
    while (1)
    {
        SBUF = 'c';        //发送字符c
        while(!TI);        //串口中断标志位
        TI = 0;
        DelayMs(240);      //延时循环发送
        DelayMs(240);
        DelayMs(240);
        DelayMs(240);
    }
}
```

使用DB9串口与电脑连接，神舟51开发板的配置如下表3-46所示：

表3-46 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
RXD	J21.RXD	跳帽	J21. RS232		单片机串口接收
TXD	J22.TXD	跳帽	J22. RS232		单片机串口发送
J3 DB9				一根母到母交叉串口线	与 PC 电脑连接
实验现象：如果看到超级终端显示的“C”字符，则说明操作成功，否则操作失败。 注意：下载程序的时候，J21、J22 的跳冒应该跳到 USB 这边，否则无法下载程序					

连接图如下图3-76所示：

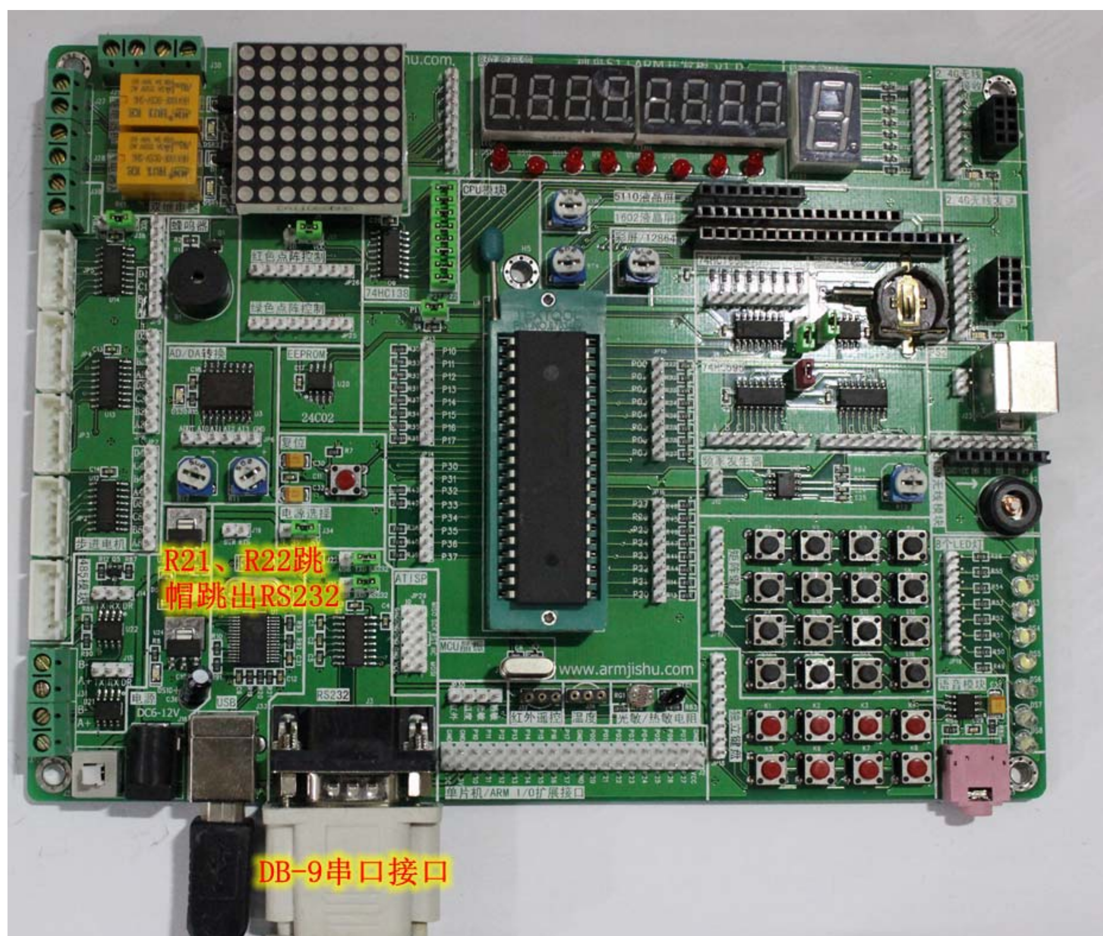


图 3-76 串口通信硬件连接实物图

知识要点：

1．本实验是通过单片机的串口向 PC 发送一个字符'c'，在 PC 的超级终端中查看串口消息。通过本例程了解单片机串口的基本原理，理解并掌握单片机的串口通信知识。

2.实验前首先使用一根母到母交叉串口线连接 51 开发板 DB9 串口与电脑，打开电脑的超级终端工具，将波特率设置为 9600，无奇偶校验，设置正确后可以看到电脑接收到的测试字符'c'如图 3-77，则说明操作成功，否则操作失败。

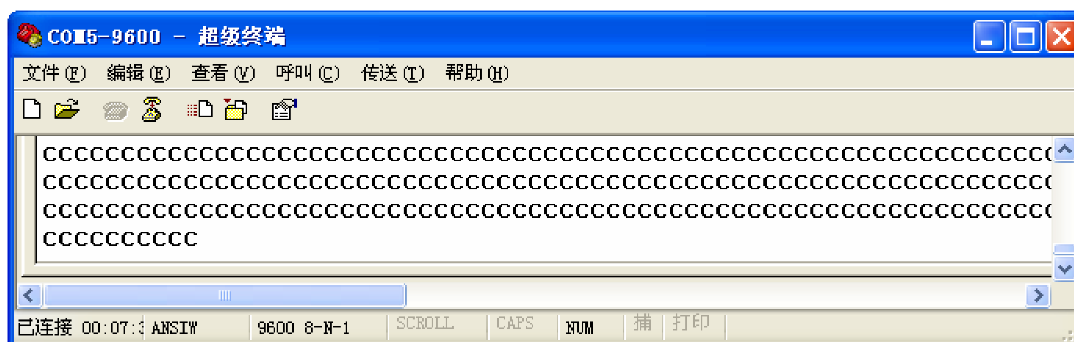


图 3-77 电脑串口终端接收到单片机发送的“C”

3．注意：晶振 11.0592MHz，波特率为 9600，使用串口或者 USB 转串口要检查 J21 和 J22 跳线跳帽的位置。

4.我们先看一下串口的初始化：

“SCON=0x50”，0x50=0101 0000;将它对的 SCON 寄存器可知：选择了工作方式 1；使

能接收。

“TMOD |= 0x20”、“TH1 = 0xFD”、“TR1 = 1 “这三条语句，设置了通信的波特率。

3.10.11 更多串口通讯例程

更多串口通讯相应的例程可通过北京航空航天大学出版社下载专区下载，如下表 3-47：

表 3-47 串口通信更多丰富例程（含详细注释和文档分析）

序号	例程功能
例程 01	DB9 串口输出一个字符
例程 02	USB 串口输出一个字符
例程 03	DB9 串口查询方式消息打印
例程 04	USB 串口查询方式消息打印
例程 05	DB9 串口 printf 打印
例程 06	USB 串口 printf 打印
例程 07	DB9 串口查询方式消息收发
例程 08	USB 串口查询方式消息收发
例程 09	DB9 串口中断方式消息收发
例程 10	USB 串口中断方式消息收发
例程 11	DB9 串口控制 LED 数码管显示
例程 12	USB 串口控制 LED 数码管显示

3.11 555 脉冲发生器

3.11.1 555 脉冲发生器的简介

在日常生活中，很多电子产品都需要脉冲。如：报警器、电子开关、电子钟表、电子玩具以及电子医疗设备等。这样就产生了众多的脉冲发生器555脉冲发生器是其中的一种。

在讲555脉冲器之前，我们先了解一下555定时器。

555定时器是将模拟电路和数字电路集成于一体的电子器件。1972年由西格尼蒂克斯公司（Signetics）研制；因内部有3个5KΩ的电阻分压器，故称555。555定时器成本低，性能可靠，只需要外接几个电阻、电容，就可以实现多谐振荡器。当然还可以实现单稳态触发器及施密特触发器等脉冲产生与变换电路。

本章节我们学习由555定时器和其他电子元件组成的多谐振荡器即：555脉冲发生器。

3.11.2 555 定时器的工作原理

555 定时器是一种模拟和数字功能相结合的中规模集成器件。一般用双极性工艺制作的称为 555，用 CMOS 工艺制作的称为 7555，除单定时器外，还有对应的双定时器 556/7556。555 定时器的电源电压范围宽，可在 4.5V~16V 工作，7555 可在 3~18V 工作，输出驱动电流

约为 200mA，因而其输出可与 TTL、CMOS 或者模拟电路电平兼容。

它的内部线路图和管脚图如下图 3-78 所示：

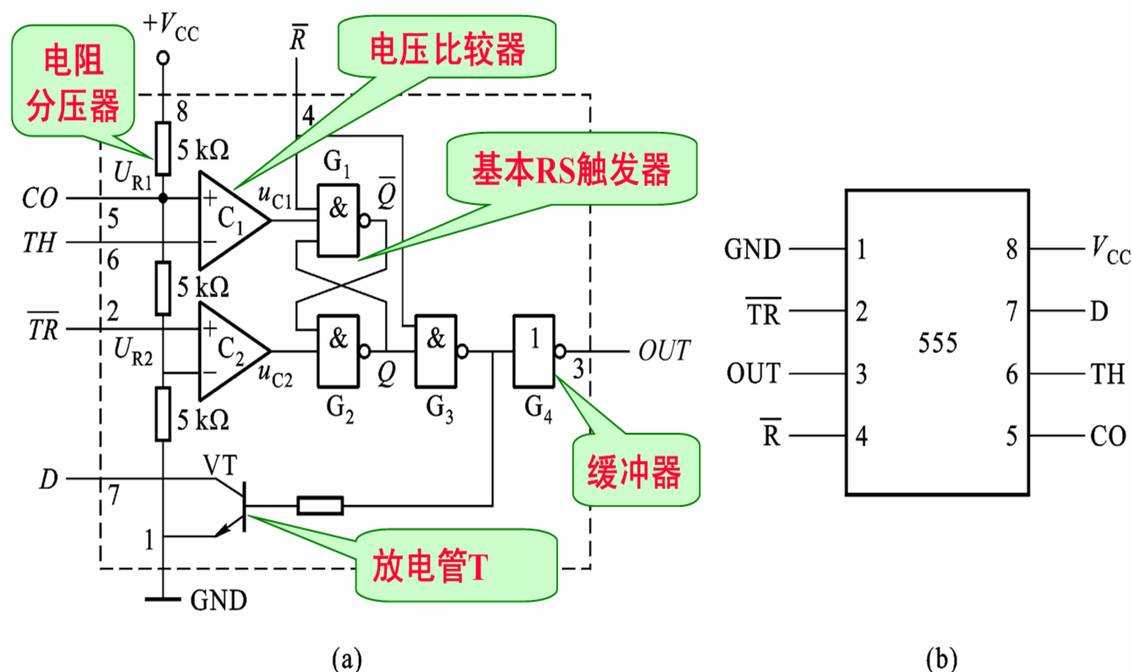


图 3-78　555 定时器内部线路图和管脚图

555 定时器的内部电路主要包括以下几部分：一个由三个相等电阻组成的分压器；两个电压比较器: A1、A2 ;一个 SR 锁存器；一个反相器和一个晶体管 T 等。

555 定时器的功能主要由两个比较器决定。两个比较器的输出电压控制 RS 触发器和放电管的状态。在电源与地之间加上电压，当 5 脚悬空时，则电压比较器 C1 的同相输入端的电压为 2VCC/3，C2 的反相输入端的电压为 VCC/3。若触发输入端 TR 的电压小于 VCC/3，则比较器 C2 的输出为 0，可使 RS 触发器置 1，使输出端 OUT=1。如果阈值输入端 TH 的电压大于 2VCC/3，同时 TR 端的电压大于 VCC/3，则 C1 的输出为 0，C2 的输出为 1，可将 RS 触发器置 0，使输出为 0 电平。

555 定时器的引脚功能如下：

1 脚：外接电源负端 VSS 或接地，一般情况下接地。

8 脚：外接电源 VCC，双极型时基电路 VCC 的范围是 4.5 ～ 16V，CMOS 型时基电路 VCC 的范围为 3～18V。一般用 5V。

3 脚：输出端 Vo

2 脚：低触发端

6 脚：TH 高触发端

4 脚：是直接清零端。当此端接低电平，则时基电路不工作，此时不论 TR、TH 处于何电平，时基电路输出为“0”，该端不用时应接高电平。

5 脚：VC 为控制电压端。若此端外接电压，则可改变内部两个比较器的基准电压，当该端不用时，应将该端串入一只 0.01μF 电容接地，以防引入干扰。

7 脚：放电端。该端与放电管集电极相连，用做定时器时电容的放电。

3.11.3 硬件原理及连接

用 555 定时器组成的多谐振荡器如下图 3-79 所示：

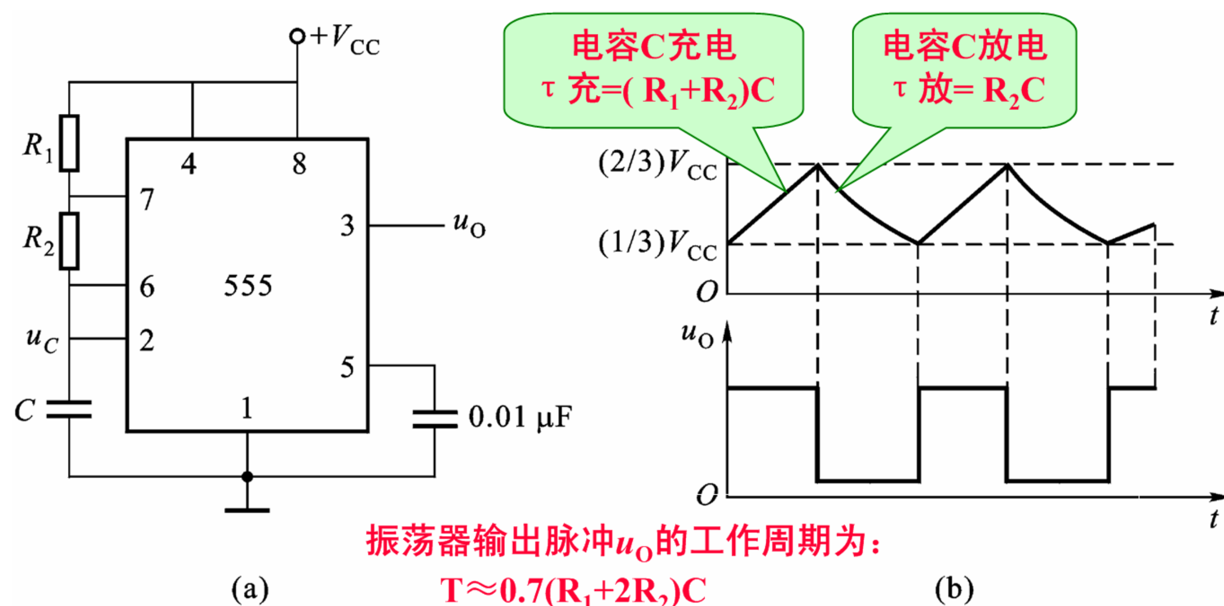


图 3-79　555 多谐振荡器

上述电路可以输出某一频率的方波，但是无法改变输出波形的频率。为了使输出波形便于调节，神舟51开发板将上图的R2改为可调电位器（滑动变阻器），通过调节电位器的阻值即可使555输出不同频率的方波。

神舟51开发板555定时器组成的多谐振荡器原理图如下图3-80所示：

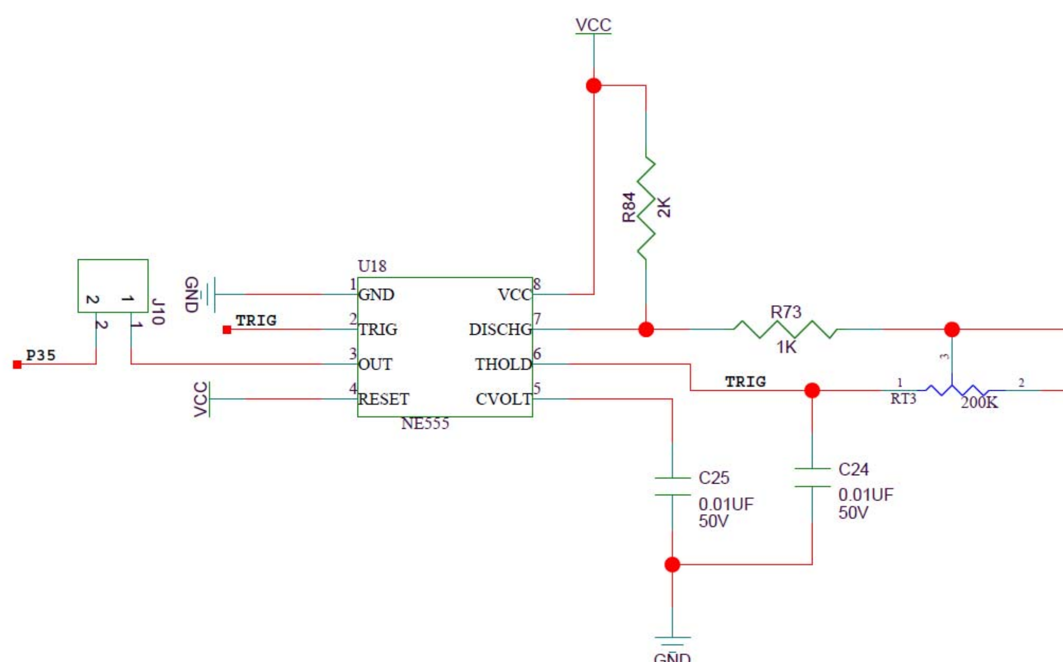


图3-80 555定时器组成的多谐振荡器原理图

3.11.4 例程 01 555 多谐振荡器蜂鸣实验

本实验使用神舟 51 开发板上 555 定时器组成的多谐振荡器，输出的脉冲驱动交流蜂鸣器。实验断开 J10 的跳线帽，使用一根杜邦线连接 J10 到 JP9 的第二个管脚。连接好以后，上电即可提到蜂鸣器鸣叫，此时旋动电位器 RT3，改变其阻值可以改变多谐振荡器的输出频率，蜂鸣器的鸣叫声会发生变化。

神舟51开发板的配置如下表3-48所示。

表3-48 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
--	J10	直连	JP9.2	1 根杜邦线	555 定时器与蜂鸣器
实验现象：上电即可提到蜂鸣器鸣叫，此时旋动电位器 RT3，改变其阻值可以改变多谐振荡器的输出频率，蜂鸣器的鸣叫声会发生变化则说明实验成功，否则实验失败。					

连接图如下图3-81所示：

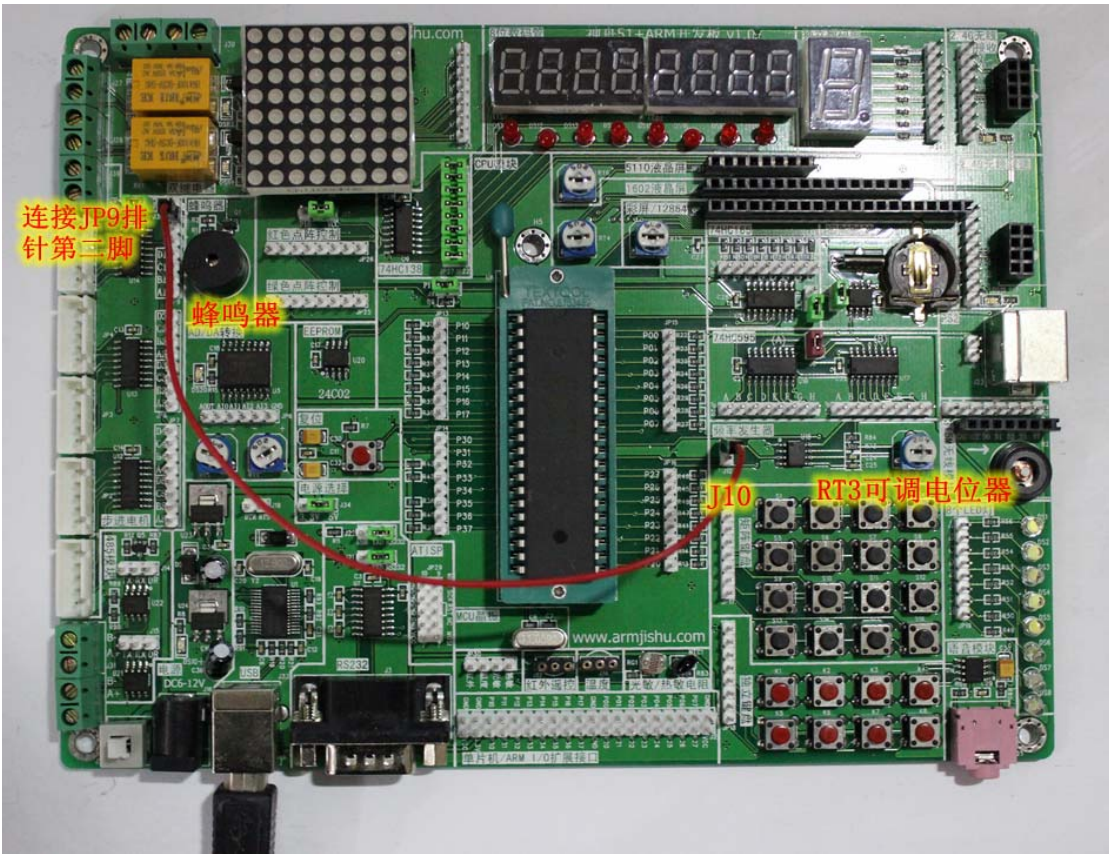


图3-81 555多谐振荡器硬件连接实物图

本实验为硬件实验，不需要程序代码。

3.11.5 更多 555 脉冲发生器例程

更多 555 脉冲发生器相应的例程可通过北京航空航天大学出版社下载专区下载，如下表 3-49：

表 3-49　555 脉冲发生器更多丰富例程介绍（含详细注释和文档分析）

序号	例程功能
例程 01	555 多谐振荡器蜂鸣实验
例程 02	1602 显示 555 脉冲发生器频率

3.12 矩阵键盘

3.12.1 矩阵按键的简介

什么是矩阵按键或者矩阵键盘？前面已经学习了独立按键,独立按键是一个按键占用单独的一个 I/O 口。当我们用到 16 个按键，或者用到更多的按键，因为 I/O 口资源是有限的、宝贵的，如果还是每个按键占用一个 I/O 口；这样明显是不合理、不现实的。这样就产生了矩阵键盘。矩阵键盘节省了 I/O 口，通常将按键排列成矩阵形式，每条水平线和垂直线在交叉处不直接连通，而是通过一个按键加以连接。

3.12.2 矩阵按键的原理与识别

那么矩阵键盘这样连接有什么优势呢？节约处理器的 I/O 口，例如如下图：总共使用了 8 个 I/O 口，如果每个 I/O 口只连一个独立按键，就可以连 8 个按键；而在矩阵式键盘中，每条水平线和垂直线在交叉处不直接连通，而是通过一个按键加以连接。这样，8 个 I/O 口就可以连成 16 个按键；而且线数越多，区别越明显，比如再多加一条线就可以构成 20 键的键盘，而直接用端口线则只能多出一键（9 键）。由此可见，在需要的键数比较多时，采用矩阵法来做键盘是合理的。

为什么 8 个 I/O 口能连成 16 个按键呢？可以通过下表查到，S1~S16 分别表示 16 个按键，A1~A4 表示 4 根 I/O 口，B1~B4 也表示 4 根 I/O 口，这样就可以让 8 个 I/O 口连接 16 个按键。如下表 3-50 所示，内部结构图与实物图如图 3-82 所示：

表 3-50 矩阵键盘按键连接关系

行/列	A1	A2	A3	A4
B1	S1	S2	S3	S4
B2	S5	S6	S7	S8
B3	S9	S10	S11	S12
B4	S13	S14	S15	S16

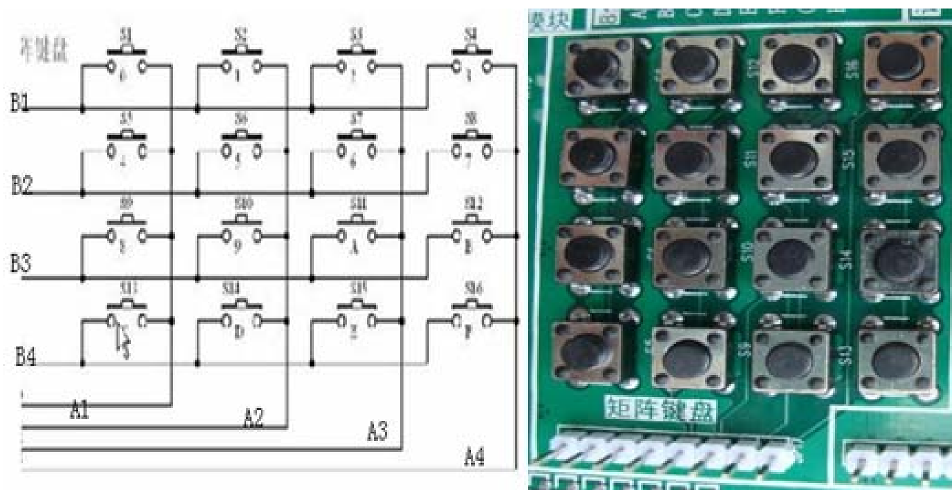


图 3-82 矩阵键盘内部结构图与实物图

3.12.3 矩阵按键的几种扫描办法，以及使用环境使用领域

常用的识别方法有行列反转扫描、行列逐级扫描、中断扫描等。

1：行列反转扫描：反转法就是通过给单片机的端口赋值两次，最后得出所按按键的值的一种算法。

2：行列逐级扫描：顾名思义，就是逐行扫描，判断是那个按键被按下，得出所按按键的值的一种算法。当然在扫描之前，可以先判断是否有按键按下。

3：中断扫描：利用外部中断或者定时器中断设计。当产生中断后，进入中断响应程序，执行扫描矩阵键盘按键函数，判断按下了那个按键。这个扫描矩阵键盘按键函数可以是行列反转扫描或者是行列逐级扫描，也可以是其他扫描方式。

3.12.4 硬件原理图

硬件连接原理图如下图 3-83 所示：

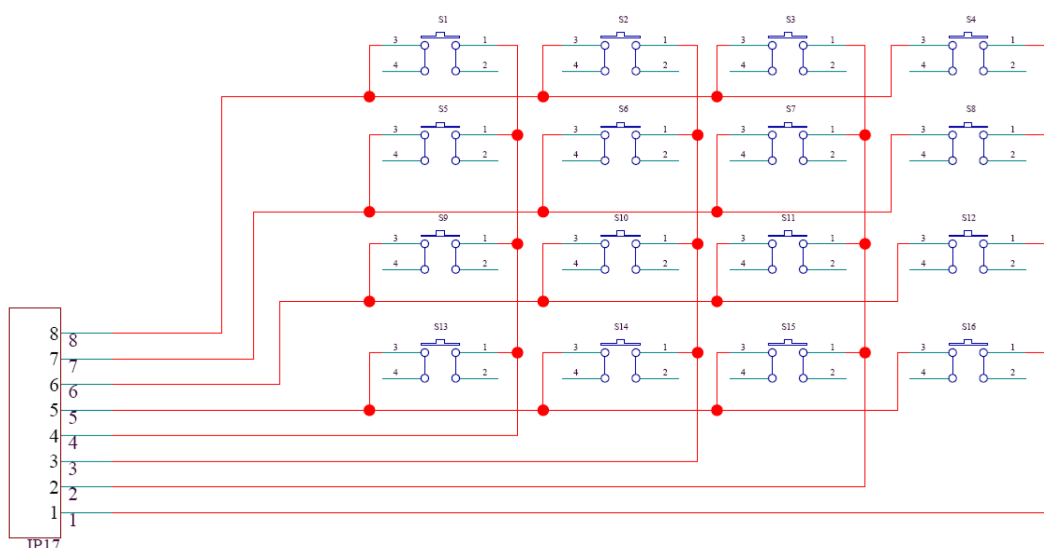


图 3-83 矩阵键盘硬件连接原理图

3.12.5 例程 01 矩阵键盘实现

代码如下：

```
/*****************************************************
* 例程：矩阵键盘按键的识别
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：按下按键数码管显示相应的内容
*****************************************************/
#include<reg52.h>
#define uchar unsigned char
#define uint unsigned int
uchar key,n;
//定义变量
uchar code table[]={0xee,0xde,0xbe,0x7e,0xed,0xdd,0xbd,0x7d,0xeb,
                    0xdb,0xbb,0x7b,0xe7,0xd7 ,0xb7,0x77};
//反转法矩阵键盘的各个按键的计算值
uchar code yin[]={0xc0,0xcf,0xa4,0xb0,0x99,0x92,0x82,0xf8,0x80,
                  0x90,0x98,0x83,0xc6,0xa1,0x86,0x8e};
 //共阳极数码管显示 0~F
void delay(uint i) //延时函数
{
    while(i--);
}
void keyscan()
{
    uchar l,h,i;   //定义局部变量，用得出低 4 位的值，用 h 得出高 4 位的值
    P1=0x0f;     //给 P1 赋值 00001111
    l=P1&0x0f;
    if(l!=0x0f)
    {
        delay(100);
        if(l!=0x0f)
        l=P1&0x0f;   //若有键按下，得出低四位的值
    }
    P1=0xf0;           //给 P1 赋值 11110000
    h=P1&0xf0;
    if(h!=0xf0)
    {
        delay(100);
        if(h!=0xf0)
        h=P1&0xf0;      //若有键按下，得出高 4 位的值
```

```
    }
    key=l+h;                    //高4位的值与低4位的值相加
    for(i=0;i<16;i++)
    {
        if(key==table[i])    //通过查表得出n的值
        n=i;
    }
}
void main()
{
    while(1)
    {
        keyscan();
        P0=yin[n];          //在数码管上显示相应的键值
    }
}
```

硬件连接关系如表 3-51 所示：

表 3-51 矩阵键盘硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P0 口	JP15(A 向左)	直连	JP24（A 向右）	1 根 8 针扁平电缆	控制共阳数码管
P1 口	JP13(A 向左)	直连	JP17（A 向右）	1 根 8 针扁平电缆	控制矩阵键盘
实验现象：下载程序后，连好杜邦线，共阳数码管显示 0，依次按下矩阵键盘上的按键，数码管对应显示 0～F。					
注意：把 J12 的跳冒，跳开。					

下面我们分析一下反转法的实现过程。

如下图 3-84 所示，取 P1 口的低四位为行线，高四位为列线。

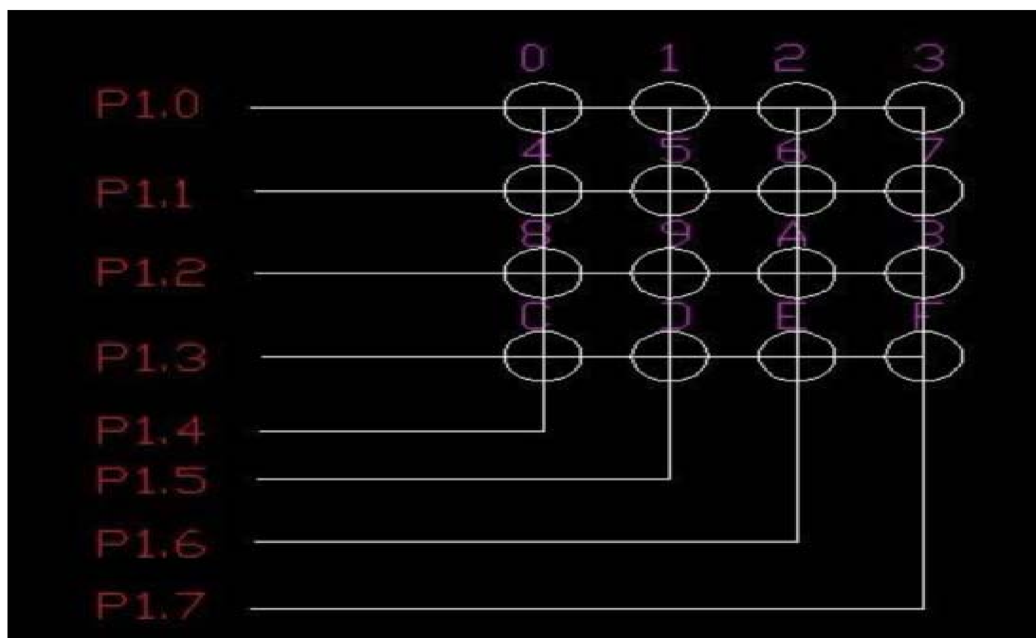


图 3-84 矩阵键盘的反转法连接

1. 我们给 P1 口赋值 0x0f，即 00001111，假设 0 键按下了，则这时 P1 口的实际为 00001110；

2. 我们给 P1 口再赋值 0xf0，即 11110000，如果 0 键按下了，则这时 P1 口的实际值为 11100000；

3. 我们把两次 P1 口的实际值相加得 11101110，即 0xee。

由此我们便得到了按下 0 键时所对应的数值 0xee，以此类推可得出其他 15 个按键对应的数值，有了数值和键盘按键的这种对应关系，矩阵键盘编程问题也就解决了，通过矩阵键盘按键反馈回来的数值判断按下了哪个按键。对应关系如下图 3-85：

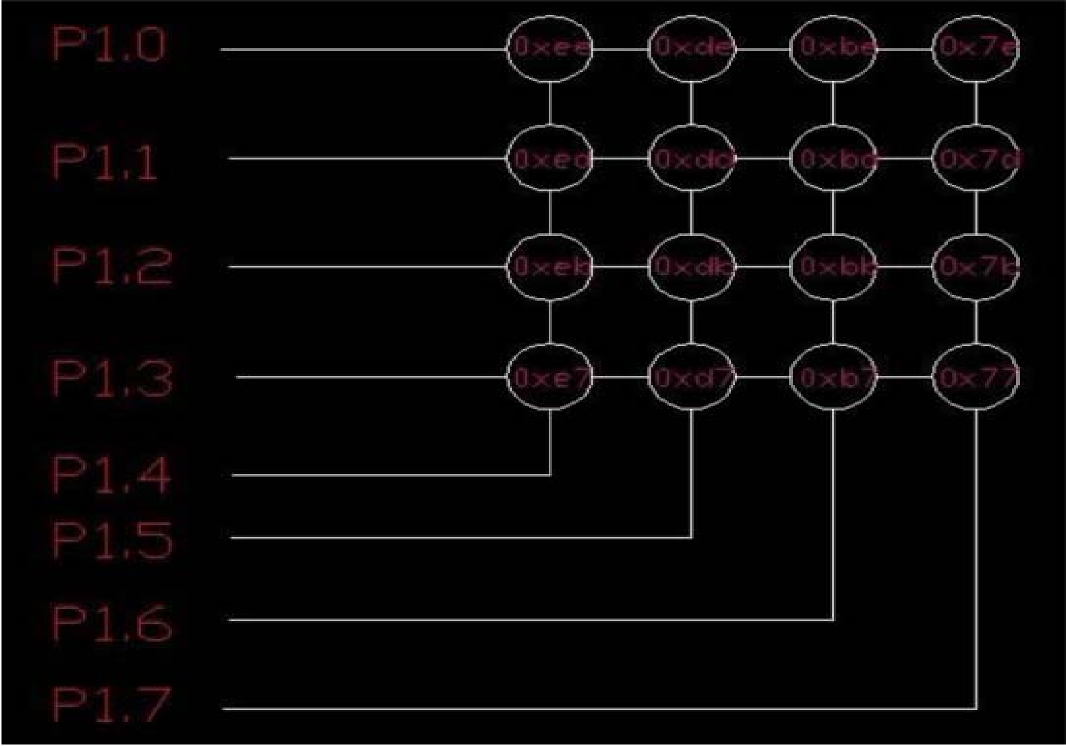


图 3-85 反转法按键对应关系

3.12.6 更多矩阵键盘例程请见表 3-49

更多矩阵键盘相应的例程可通过北京航空航天大学出版社下载专区下载，如下表 3-52：

表 3-52 矩阵键盘更多丰富例程介绍（含详细注释和文档分析）

序号	例程功能
例程 01	矩阵键盘实现
例程 02	矩阵键盘行列扫描

3.13 串转并扩展(HC595)

3.13.1 74HC595 的简介

1）HC595 的出现背景

众所周知，I/O 口资源是非常宝贵的。特别是在做复杂一点的系统的时候，I/O 口资源可能会出现不够用的情况。为此各个厂商分别设计很多芯片来解决这个问题。74HC595 就是其中的一种芯片。

2）HC595 的特性

74HC595 是美国国家半导体公司生产的通用移位寄存器芯片。它是硅结构的 CMOS 器件，兼容低电压 TTL 电路，具有较强的负载能力；而被广泛应用于 MCU(微控制器)、MPU（微处理器）的 I/O 口扩展。尤其在电子显示屏制作当中被广泛的应用。而且它的市场价格非常低廉，就目前而言，平均每片单价 8 毛钱左右。

3.13.2 串转并扩展(HC595)的工作原理

74HC595 所实现的功能： HC595 是带锁存功能的三态输出的 8 位串行输入/并行输出的移位寄存器。由于它自带锁存器，所以其数据在移位寄存器中的移位与锁存器的输出时独立的。当数据移位时，可以保存锁存器输出的数据不改变，等所有 8 位数据全部串行输入完成移位操作后，一次性的将数据打入锁存器中，从而实现了并行输出的同步改变。另外该芯片可以进行级联，能够实现多个并口扩展。

74HC595 芯片的管脚功能图如下图 3-86 所示：

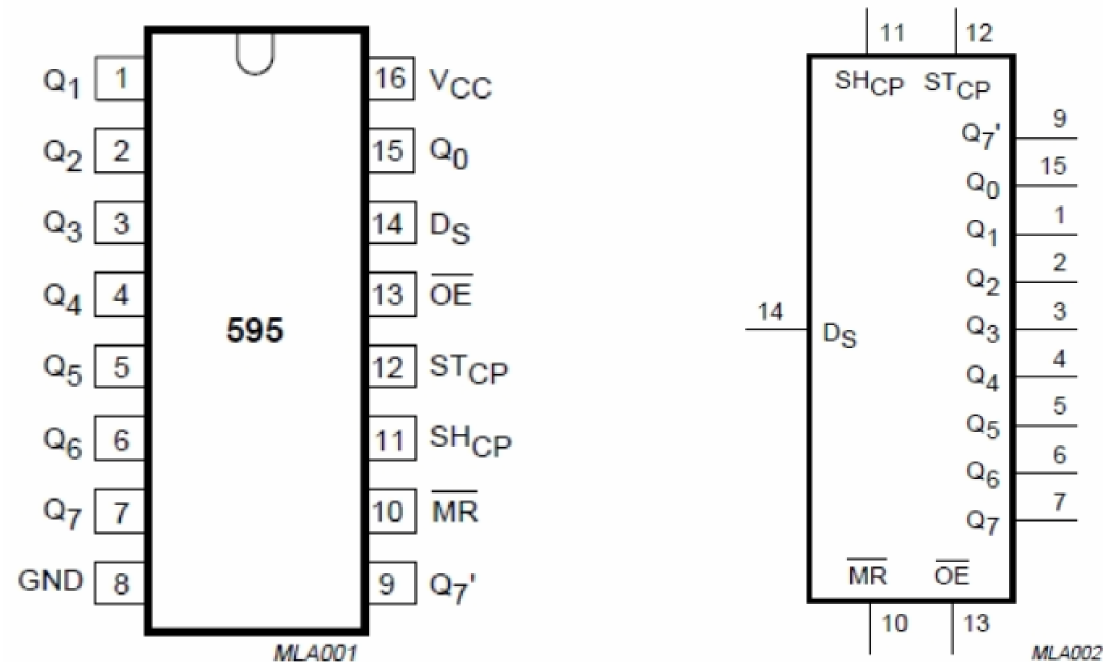


图 3-86　HC595 管脚图与 HC595 功能图

管脚功能说明：

1） Q1～Q7 （1、2、3、4、5、6、7、15 引脚）三态输出管脚
2） GND （8 引脚） 电源地
3） Q7’ （9 引脚） 串行数据输出管脚
4） /MR （10 引脚） 移位寄存器清零端
5） SHcp （11 引脚） 数据输入时钟线
6） STcp （12 引脚） 输出存储器锁存时钟线
7） /OE （13 引脚） 输出使能
8） DS （14 引脚） 数据线
9） VCC （16 引脚） 电源端

讲到这里我们会有这样的疑问，那我们怎么使用这个芯片呢？光看引脚功能说明是不行的，下面我们看一下 HC595 的逻辑功能表 3-53：

表 3-53 HC595 的逻辑功能表

输入					输出		功能
SHcp	STcp	OE	MR	Ds	Q7	Qn	
×	×	L	↓	×	L	NC	MR为低电平时仅仅影响移位寄存器
×	↑	L	L	×	L	L	空移位寄存器到输出移位寄存器
×	×	H	L	×	L	Z	清空移位寄存器，并行输出为高阻态
↑	×	L	H	H	Q6	NC	逻辑高电平移入移位寄存器状态0，包含所有的移位寄存器状态移入，例如，以前的状态6（内部Q6’）出现在串行输出位
×	↑	L	H	×	NC	Qn’	移位寄存器的内容到达保持寄存器并从并口输出
↑	↑	L	H	×	Q6	Qn’	移位寄存器内容移入，先前的移位寄存器的内容到达保持寄存器并输出

从 74HC595 的逻辑功能表中我们可以分析出 74HC595 的工作过程：

数据的串入和内部数据移位的操作由 SHcp（11 引脚）引脚控制。SHcp 引脚提供一个上升沿将移位寄存器中的数据由 Q0 向 Q7 依次移动一位，同时将数据线上的电平打入 Q0，而最高位的数据 Q7 从 Q7’端移出。如果把 Q7’与另一片 74HC595 的数据端连接，那么 Q7’的串行输出就是第 2 片 74HC595 的串行数据输入，从而实现级联。

74HC595 在移位的过程中并不影响其锁存器的输出，移位寄存器中的数据是通过锁存端的上升沿打入到锁存器中的。正是由于 74HC595 具备了锁存功能，因而可以保证并行输出数据的稳定和数据同步改变的功能。

74HC595 基本工作步骤：假设送 8 位数到 74HC595,并行输出。

第一步：目的：将 8 位数据移入 74HC595，即数据的串入。

方法：将数据按位发送到 74HC595 的输入端 DS，给 SHcp 一个上升沿，DS 上的数据位被移入 74HC595 中。循环 8 次，逐位送入。

第二步：目的：并行输出数据，即数据并出。

方法：给 STcp 一个上升沿。

3.13.3 硬件原理与连接

硬件连接原理图如下图 3-87、3-88 所示：

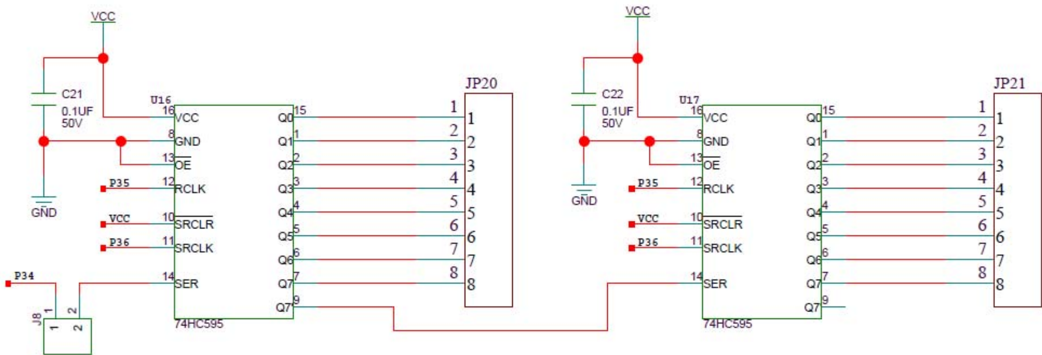


图 3-87 串转并扩展(HC595)硬件连接原理图

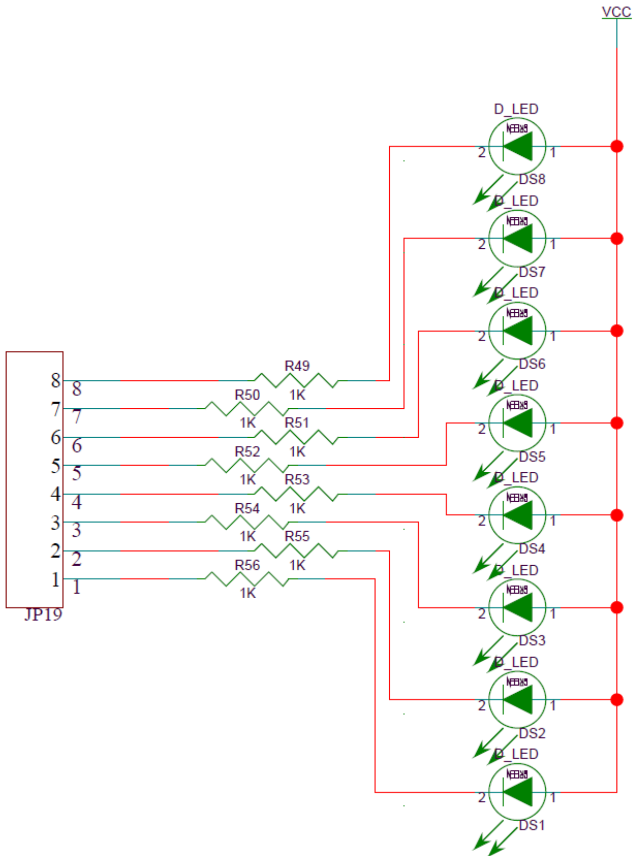


图 3-88 LED 硬件连接电路

硬件连接实物图如下图 3-89 所示：

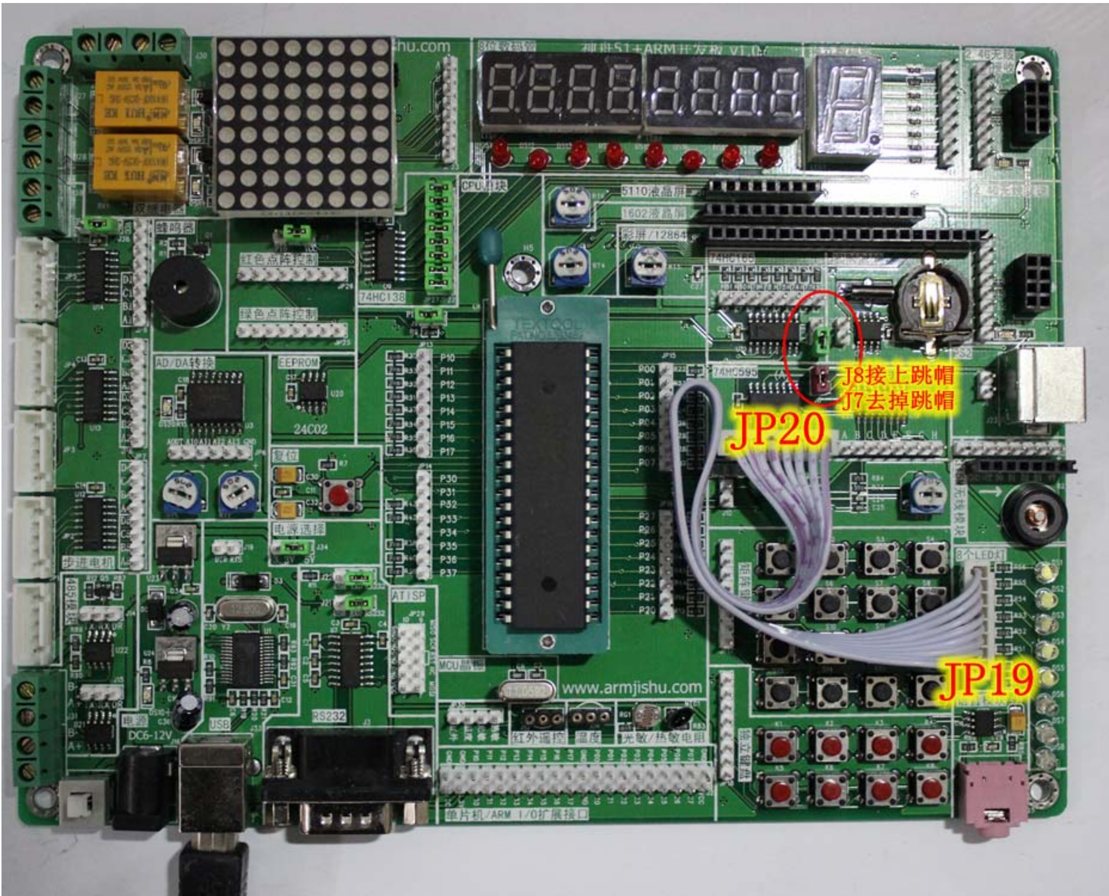


图 3-89 硬件连接实物图

硬件连接对应关系如表 3-54 所示：

表 3-54 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P0	JP20（A 向下）	直连	JP19(A 向右）	8 位排线	74HC595(A)控制 LED 灯
用跳帽短接 J8（连接 P3.4），注意必须拔掉 J7 的跳帽					

3.13.4 例程 01 74HC595 控制多个LED灯点亮

代码如下：

```
/*******************************************************
* 例程：74HC595 控制多个 LED 灯点亮
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：74HC595 控制多个 LED 灯点亮
*******************************************************/
#include <reg52.h>//包含头文件，一般情况不需要改动，
                //头文件包含特殊功能寄存器的定义
#include <intrins.h>   //包含头文件
```

```
sbit LATCH = P3^5;   //锁存信号输入
sbit SRCLK = P3^6;   //串行数据时钟输入
sbit SER     = P3^4;   //串行数据输入
/*------------------------------------------------
 uS 延时函数，含有输入参数 unsigned char t，无返回值
 unsigned char 是定义无符号字符变量，其值的范围是
 0~255 这里使用晶振 12M，精确延时请使用汇编,大致延时
 长度如下 T=tx2+5 uS
------------------------------------------------*/
void DelayUs2x(unsigned char t)
{
 while(--t);
}
/*------------------------------------------------
 mS 延时函数，含有输入参数 unsigned char t，无返回值
 unsigned char 是定义无符号字符变量，其值的范围是
 0~255 这里使用晶振 12M，精确延时请使用汇编
------------------------------------------------*/
void DelayMs(unsigned char t)
{
 while(t--)
 {
     //大致延时 1mS
     DelayUs2x(245);
     DelayUs2x(245);
 }
}
/*------------------------------------------------
          发送字节程序
------------------------------------------------*/
void SendByte(unsigned char dat)
{
    unsigned char i;
    for(i=0;i<8;i++)
    {
        SRCLK=0;          //先将串行时钟输入端 SRCLK 置成低电平
        SER=dat&0x80;     //每次只取一位
        dat<<=1;          //每次只取一位进行左移
        SRCLK=1;          //数据在串行时钟输入端 SRCLK 的上升沿输入到移位寄存器中
    }
}
/*------------------------------------------------
           595 锁存程序
     595 级联发送数据后，锁存有效
```

```
-----------------------------------------------*/
void Out595(void)
{
            LATCH=0;   //锁存
            _nop_();       //空指令
            LATCH=1;   //LATCH 上升沿并行数据输出
}
/*-----------------------------------------------
                主函数
-----------------------------------------------*/
main()
{
      unsigned char Led=0xff;         //初始化 led 值  1111 1111
      SendByte(0xff);                 //初始化 595 使他为高电平  让 LED 处于熄灭状态
      Out595();                       //595 锁存并行输出
      while(1)                        //主循环
      {
            Led = 0xaa;               //Led 赋值，0xaa 为 1010 1010
            SendByte(Led);            //调用 595 驱动程序  把 LED 的数据送到 595
            Out595();                 //595 锁存并行输出
            DelayMs(500);             //延时
      }
}
```

硬件连接对应关系如表 3-55 所示：

表 3-55 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P0	JP20（A 向下）	直连	JP19(A 向右)	8 位排线	74HC595(A)控制 LED 灯
用跳帽短接 J8（连接 P3.4），注意必须拔掉 J7 的跳帽					
实验现象：下载程序后，我们可以看到 1 个 LED 灯部分点亮					

知识要点：

1.通过 595 输入数据分以下三步：

第一步：将要准备输入的位数据移入到 74HC595 数据输入端上，送数据到串行数据输入 SER，即 P3.4 管脚，我们可以对应程序代码中的说明看得到。

第二步：将数据逐位移入到 74HC595 中，即 SRCLK 所连接的 P3.6 产生一个上升沿，可以将 P3.4 管脚上的数据移入到数据寄存器中。

第三步：并行输出数据，即数据并出；输出锁存器控制脉冲所对应管脚 P3.5，产生一个上升沿，将由 P3.4 管脚上已移入的数据(在 74HC595 芯片的数据寄存器中)送入到 74HC595 的输出锁存器。

2.在原理图中可以看到，我们只需要用跳帽短接 J8（连接 P3.4）就可以了，因为其他管脚在电路板上本身就是已经连上了

3.595 芯片上的 Q0-Q7 总共 8 根线（JP20）与 8 个 LED（JP19）通过排线相连接好，也就是说，Q0-Q7 任意一个管脚都是单独驱动一个 LED 灯的。

4.最后输出的数据，在 while 循环里，串行数据输入通过 595 转为并行数据输出，点亮一半的 LED 灯，一直循环。

3.14 并转串扩展（HC165）

3.14.1 并转串扩展 74HC165 简介

我们在设计单片机系统时，外部设备接入单片机会占用 I/O 口。比如 8 个独立按键，直接和单片机相连，会占用了 8 个管脚。以 51 单片机为例，它的 I/O 口只有 32 个，我们做复杂一点的系统的时候，可能会出现管脚不够用的情况。而 8 个独立按键就占用了 8 个管脚，显然很浪费。这个时候，用集成芯片 HC165 将数据信息转成串行的送入单片机，可以帮助我们省管脚。

这时候您可能会有这样的疑问：串转并扩展 HC595 和并转串扩展（HC165）都起到节约 I/O 管脚的作用，那它们的区别是什么呢？细心的话您就会发现：用串转并扩展 HC595 时主要是单片机往外设发送数据，而用并转串扩展 HC165 时是单片机接收外设的数据。

3.14.2 并转串扩展（HC165）的工作原理

74HC165 是一种并入串出的芯片，它是一款高速 CMOS 器件，兼容低电压 TTL 电路。其功能为并行输入串行输出的驱动器。并且支持并转串扩展。

74HC165 的管脚图和功能图如下图 3-90 所示：

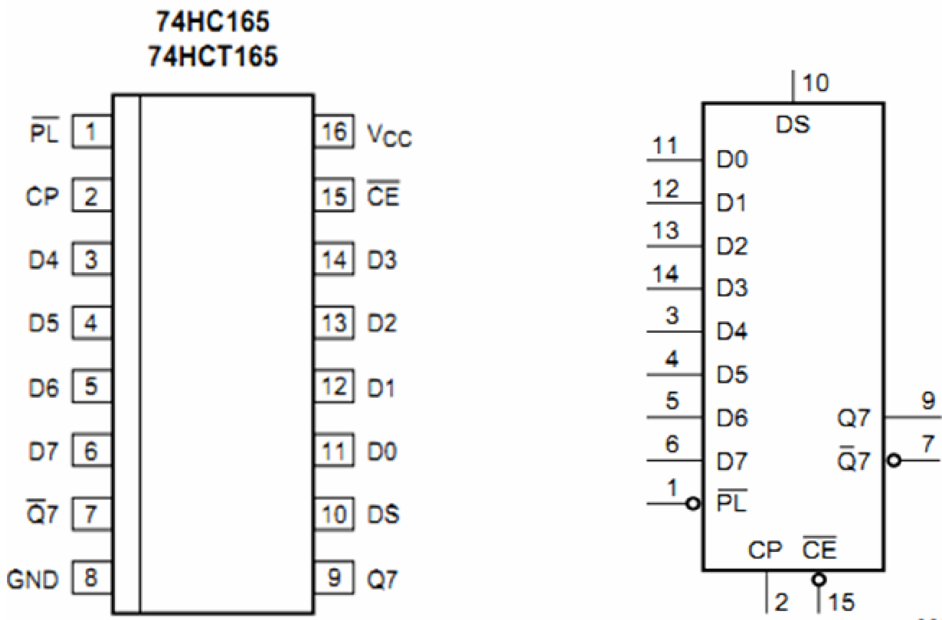


图 3-90 HC165 管脚图与 HC165 功能图

管脚功能如下表 3-56 所示：

表 3-56 HC165 管脚功能介绍

序号	引脚标号	引脚号	功能
1	/PL	1 引脚	移位/置位控制端（低电平有效）
2	CP	2 引脚	时钟输入端（上升沿有效）
3	/CE	15 引脚	时钟输入端（上升沿有效）
4	D0～D7	3～6、11～14 引脚	并行数据输入端
5	Q7	9 引脚	输出端
6	/Q7	7 引脚	互补输出端
7	GND	8 引脚	地
8	DS	10 引脚	串行数据输入端
9	VSS	16 引脚	电源

光看管脚说明是不能了解芯片是怎样使用的，下面我们分析一下 HC165 是如何使用的：

1） 当输入端（SH/LD）为低：

从 D0 到 D7 口输入的并行数据将被异步地读取进寄存器内，然后通过 DS 输出

2）当输入端（SH/LD）为高：

数据将从 DS（10 引脚）输入端串行进入寄存器，在每个时钟脉冲的上升沿向右移动一位。利用这种特性，只要把 Q7 输出绑定到下一级的 DS 输入，即可实现并转串扩展。（在我们的开发板，DS 端是悬空的），通过时钟管脚，上升沿有效，在每个时钟脉冲的上升沿向右移动一位（D0 → D1 → D2 → D3……→ D7）一位一位将数据往外读出和移出。

3.14.3 硬件原理与连接

硬件连接原理图如下图 3-91 所示：

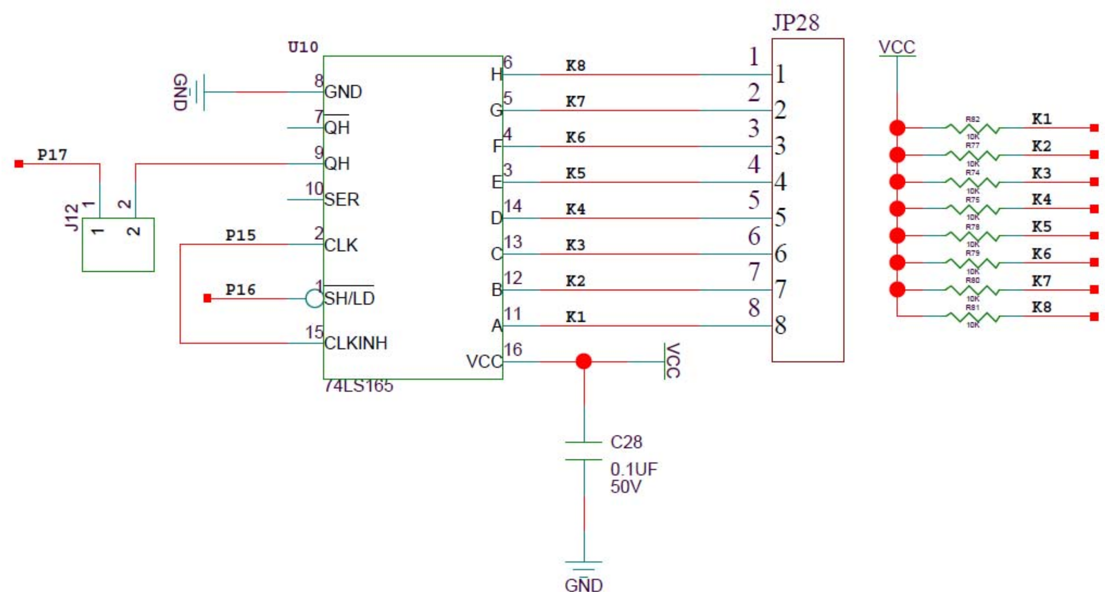


图 3-91 硬件连接原理图

先用跳帽短接 J12，用一根 8 位排线将板上的 JP28 插针的与 JP18 插针连接,再用一根 8 位排线将板上的 JP16 插针的与 JP19 插针连接,以下所以并行转串行数据实验均按此线路连接。

3.14.4 例程 01 74HC165 读按键功能 1

代码如下：

```
/****************************************************
* 例程：74HC165 读按键功能 1
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：74HC165 读按键功能，按键按下点亮相应的 LED 灯，松开熄灭
****************************************************/
#include <reg52.h>    //包含头文件，一般情况不需要改动，
                      //头文件包含特殊功能寄存器的定义
#include <intrins.h>   //包含头文件
#define   NOP()    _nop_()    /*  定义空指令  */
sbit    CLK     = P1^5;       //串行时钟
sbit    IN_PL   = P1^6;       //把数据加载到锁存器中
sbit    IN_Dat = P1^7;        //数据通过 P1.7 脚移进单片机内处理
unsigned char bdata Key;
sbit    K0=Key^0;      //位定义
sbit    K1=Key^1;      //位定义
sbit    K2=Key^2;      //位定义
sbit    K3=Key^3;      //位定义
sbit    K4=Key^4;
sbit    K5=Key^5;
sbit    K6=Key^6;
sbit    K7=Key^7;
unsigned long ReHC74165(void);   //函数声名
/*------------------------------------------------
                  主函数
------------------------------------------------*/
 main()
{
     while(1) //主循环
     {
          unsigned long   Input=ReHC74165();   //调用 165 驱动程序
          Key=Input&0xff;      //将数据传给位变量
          P2 = 0xff;           //清除

          if(K0) { P2 = 0x7f;    } //实现脉冲输入
          if(K1) { P2 = 0xbf;    } //K1  为 1 时点亮第 2 个灯
```

```
            if(K2) { P2 = 0xdf;     }
            if(K3) { P2 = 0xef;     }
            if(K4) { P2 = 0xf7;     }
            if(K5) { P2 = 0xfb;     }
            if(K6) { P2 = 0xfd;     }
            if(K7) { P2 = 0xfe;     }
        }
    }
/*-----------------------------------------------
              接收串行数据程序
此部分为 74HC165 的驱动程序使用 SPI 总线连接
------------------------------------------------*/
unsigned long ReHC74165(void)
{
    unsigned char i;
    unsigned int indata;
      IN_PL=0;
      NOP();                          //短暂延时 产生一定宽度的脉冲
      IN_PL=1;                        //将外部信号全部(并行的 8 位都将读进来)读入锁存器中
      NOP();
      indata=0;                        //保存数据的变量清 0
      for(i=0; i<8; i++)
        {
          indata=indata<<1;                  //左移一位
          if(IN_Dat==1)indata=indata+1;   //如果 IN_Dat 检测到高电平保存数据的变量加 1
          CLK=0;     //时钟置 0
          NOP();
          CLK=1;     //时钟置 1
        }
      return(~indata);  //将保存数据的变量取反后返回
}
```

硬件连接对应关系如表 3-57 所示：

表 3-57 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P2	JP16（A 向右）	直连	JP19(A 向左)	8 位排线	74HC164 接收按键信号
	JP28（A 向下）	直连	JP18(A 向右)	8 位排线	单片机控制 LED 灯
用跳帽短接 J12					
实验现象：下载程序后，我们按下独立按键后，对应的 LED 灯点亮；松开后熄灭					

硬件连接实物图如下图 3-92 所示：

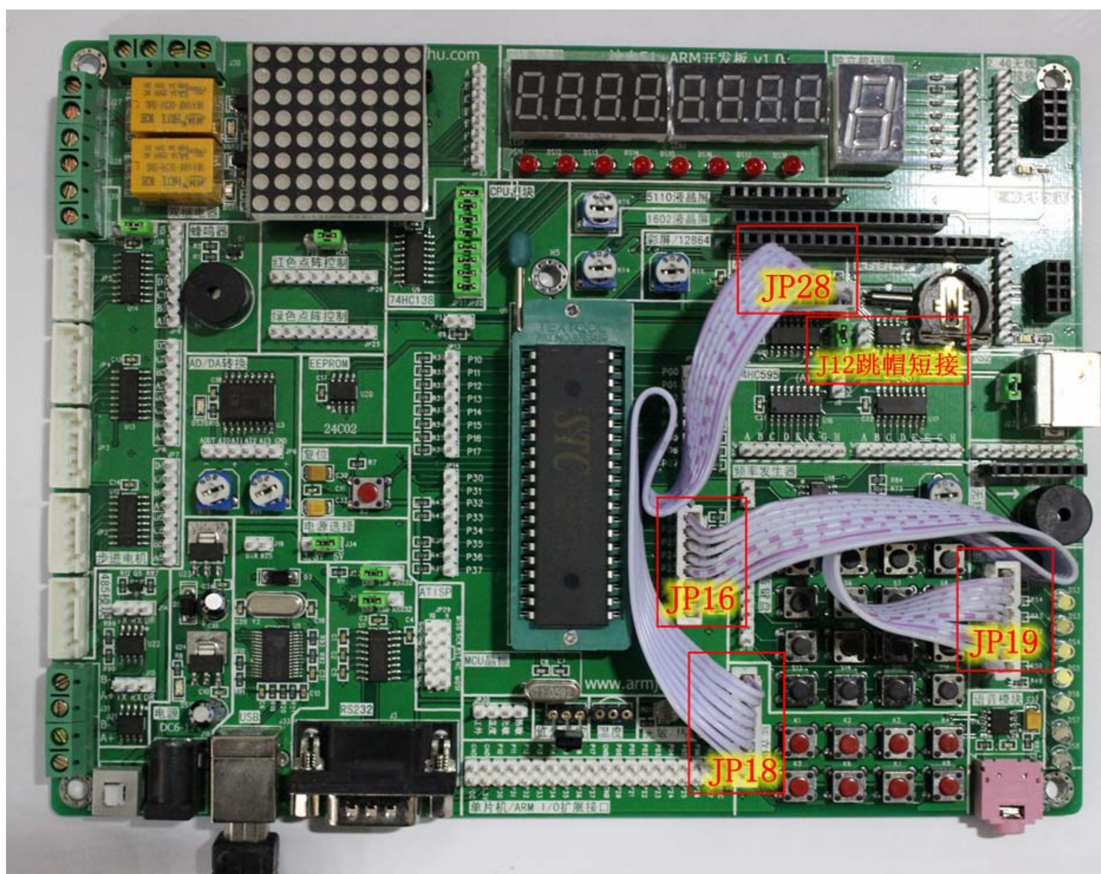


图 3-92 硬件连接实物图

知识要点：

1.并行数据输入通过 HC165 转为串行数据输出送给单片机，单片机点亮相应的 LED 灯。

2. bdata 用于将变量定义在可位寻址片内数据存储区，允许位与字节混合访问；通过语句“unsigned char bdata Key;”位变量 Key 被定义为 bdata 存储类型，编译时编译器将把该变量定位在 51 单片机片内数据存储区（RAM）中的位寻址区，即可以用 bit 的方式去访问。

3.具体代码分析，在程序中：

第一步是读入数据，通过 PL 时钟管脚，当 PL 输入为低时，从 D0 到 D7 口输入的并行数据将被异步地读取进寄存器内；而当 PL 为高时，数据将从 DS 输入端串行进入寄存器。通过我们的第一步操作，外部并行的数据已经全部(并行的 8 位都将读进来)读入 165 芯片里了。涉及的主要代码如下：

“IN_PL=0;　NOP();　IN_PL=1;　NOP();”

第二步是读串行数据出来

通过 CP 和 CE 时钟管脚，上升沿有效，在每个时钟脉冲的上升沿向右移动一位(D0 → D1 → D2 → D3……→ D7 逐渐一位一位往外读出和移出）。

涉及的主要代码：“if(IN_Dat==1)indata=indata+1;”

“CLK=0;　NOP();　CLK=1;”

第三步：因为每次并行口都是输入 8 位，所以我们增加了一个 for 循环，来每次取 8 位数据，刚好取完并行输入一次的所有数据，然后来进行分析：

“for(i=0; i<8; i++)”

5.最后因为通过 74HC165 芯片过来的数据都是跟并行读到的信号电平是一致的，那么当我们某个按键按下的时候，是低电平有效，所以要使得按下按键的对应键亮，ReHC74165()

函数通过返回一个取反的值，即原本按下键对应管脚为低电平 0，返回的取反值就变成 1 了，所以通过在 main()函数的逐个管脚用 if(keyx)来判断，就可以判断出哪个按键被按下了，从而显示对应的 LED 灯。

3.15 译码实验（HC138）

3.15.1 什么是译码器

使用单片机控制外部设备时，一根引脚只能控制一个设备的变化，单控制多个设备时，我们就需要多根引脚去控制。因为主芯片的管脚数量有限，主芯片的成本非常高，为了能够管理更多的终端按键和设备，就发明了成本比较低的译码器来扩充主芯片的管脚数

译码是编码的逆过程，在编码时，每一种二进制代码，都赋予了特定的含义，即都表示了一个确定的信号或者对象。把代码状态的特定含义“翻译”出来的过程叫做译码，实现译码操作的电路称为译码器。或者说，译码器是可以将输入二进制代码的状态翻译成输出信号，以表示其原来含义的电路。

3.15.2 译码器的实现原理

下面图 3-93 收译码器的逻辑电路图，A0、A1、A2 三个二进制灵活组合，可以变成 8 个不同的数字，这个是二进制的基础知识，比如二进制 000 等于十进制的 0，二进制 001 等于十进制的 1，二进制 010 等于十进制的 2，二进制 011 等于十进制的 3……以此类推，总共是 8 个不同的数字。

下面的这个逻辑电路，有兴趣的可以自己推推看，看看是不是 A0、A1、A2 灵活变换数字，可以有规律的控制 Y0~Y7 这 8 根管脚。

其中 E1、E2、E3 是灵活控制这个译码器 3 个控制管脚，下面还有张表 3-58，可以看到不同状态的对应表格的数据；有可能不同厂家的译码器各种功能有不同，但大概原理都是这样的，有兴趣的可以自行研究和推推看这个逻辑是否正确。

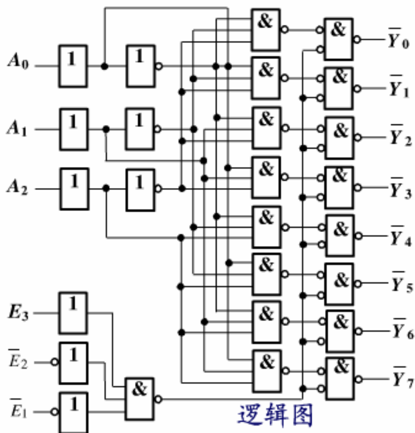


图 3-93 译码器逻辑电路图

表 3-58 138 译码器逻辑功能

输入					输出							
E1	$\overline{E_{2A}}+\overline{E_{2B}}$	A2	A1	A0	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
×	1	×	×	×	1	1	1	1	1	1	1	1
0	×	×	×	×	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	1	1	1
1	0	0	1	0	1	1	0	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1	1	1
1	0	1	0	0	1	1	1	1	0	1	1	1
1	0	1	0	1	1	1	1	1	1	0	1	1
1	0	1	1	0	1	1	1	1	1	1	0	1
1	0	1	1	1	1	1	1	1	1	1	1	0

3.15.3 HC138 译码器芯片介绍

74LS138 是目前常用的三线——八线译码器（变量译码器），它有三根输入线，可以输入三位二进制数码，共有八种状态组合，即可译出 8 个输出信号。管脚图与逻辑图如图 3-94 所示。74LS138 的功能见上表 3-58。

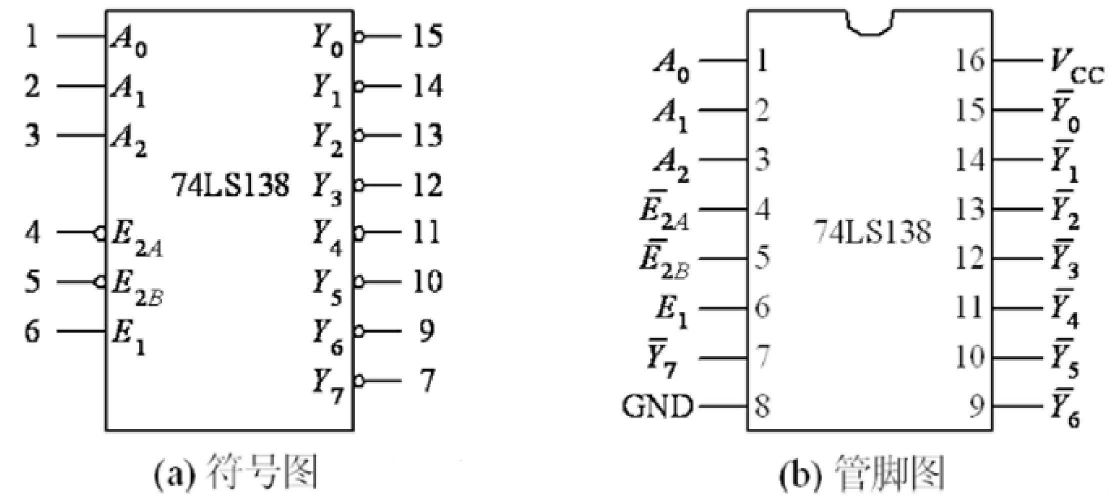


图 3-94 译码器管脚图与逻辑图

74HC138 是译码器中常用的一种，它可接受 3 位二进制加权地址输入（A0, A1 和 A2），并当使能时，提供 8 个互斥的低有效输出（Y0 至 Y7）。74HC138 特有 3 个使能输入端：两个低有效（E1 和 E2）和一个高有效（E3）。除非 E1 和 E2 置低且 E3 置高，否则 74HC138 将保持所有输出为高。利用这种复合使能特性，仅需 4 片 74HC138 芯片和 1 个反相器，即可轻松实现并行扩展，组合成为一个 1-32（5 线到 32 线）译码器。任选一个低有效使能输入端作为数据输入，而把其余的使能输入端作为选通端，则 74HC138 亦可充当一个 8 输出多路分配器，未使用的使能输入端必须保持绑定在各自合适的高有效或低有效状态。

实验中我们通过译码器(HC138)来控制多个外设管脚，使得这些管脚不会同时使用，这样可以用 3 个 I/O 口控制 8 个 I/O 口，控制 8 个芯片；当然其他还有很多作用，学会这颗芯片之后，可以灵活的加以应用。

输入 3 只脚是 3 位二进制数，最大是 111 也就是 8；输出是 8 个脚，根据输入的二进制数来输出，如果输入是 101 那么就是第 5 只脚低电平，表示十进制数是 5。其实 3-8 译码器的功能就是把输入的 3 位二进制数翻译成十进制进行输出。

3.15.4 硬件原理与连接

硬件连接原理图如下图 3-95 所示：

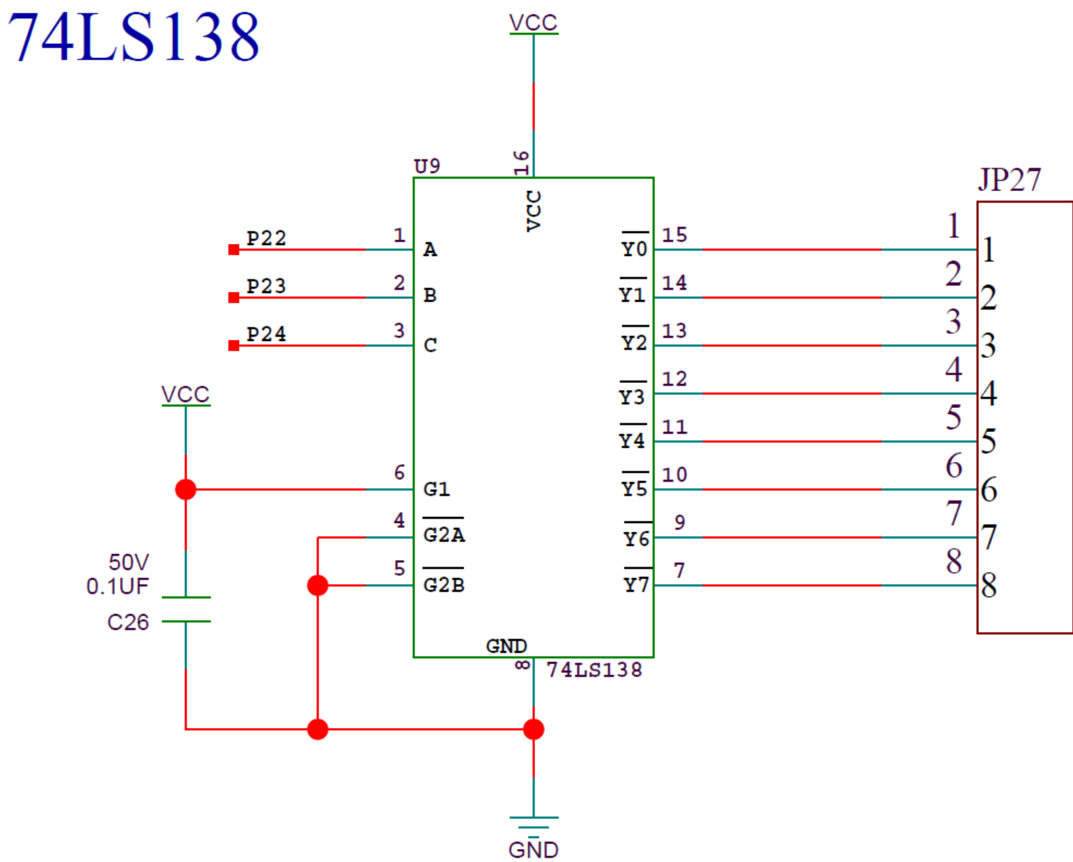


图 3-95 138 译码器原理图

可以看到原理图中，G1 连着 VCC，而 G2A 和 G2B 连着 GND；即之前讲解的 G1 就是 E1，G2A 和 G2B 就是 E2A 和 E2B，可以看下表，在神舟 51+ARM 的开发板中，已经默认把译码器设置为有效了。

然后由 P22，P23，P24 三个管脚来控制 A0~A2 来管理 Y0~Y7 8 个输出管脚，具体输出可以参见下表 3-59：

表 3-59 74LS138 的逻辑功能

输入					输出							
E1	$\overline{E_{2A}}+\overline{E_{2B}}$	A2	A1	A0	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
×	1	×	×	×	1	1	1	1	1	1	1	1

0	×	×	×	×	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	1	1	1
1	0	0	1	0	1	1	0	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1	1	1
1	0	1	0	0	1	1	1	1	0	1	1	1
1	0	1	0	1	1	1	1	1	1	0	1	1
1	0	1	1	0	1	1	1	1	1	1	0	1
1	0	1	1	1	1	1	1	1	1	1	1	0

3.15.5 例程 01 三八译码器点亮 1 个LED灯

代码如下：

```
/*******************************************************
* 例程：三八译码器点亮 1 个 LED 灯
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：三八译码器控制点亮 1 个 LED 灯
*******************************************************/
#include <reg52.h>        //包含头文件，头文件包含特殊功能寄存器的定义
sbit HC138A = P2^2;     //定义译码器输入端 A 在 P2.2 管脚上
sbit HC138B = P2^3;     //定义译码器输入端 B 在 P2.3 管脚上
sbit HC138C = P2^4;     //定义译码器输入端 C 在 P2.4 管脚上
/*-----------------------------------------------
                    主函数
-----------------------------------------------*/
void main (void)
{
    // 点亮第一个 LED 灯
    HC138C = 0; HC138B = 0; HC138A = 0;     //译码器输入 000,输出为 1111 1110
}
```

硬件连接对应关系如表 3-60 所示：

表 3-60 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
	JP27（A 向右）	直连	JP19（A 向左）	1 根 8 位排线	译码器控制单个 LED 灯
实验现象：下载程序后，我们可以看到 1 个 LED 灯点亮					

效果实物图如图 3-96 所示：

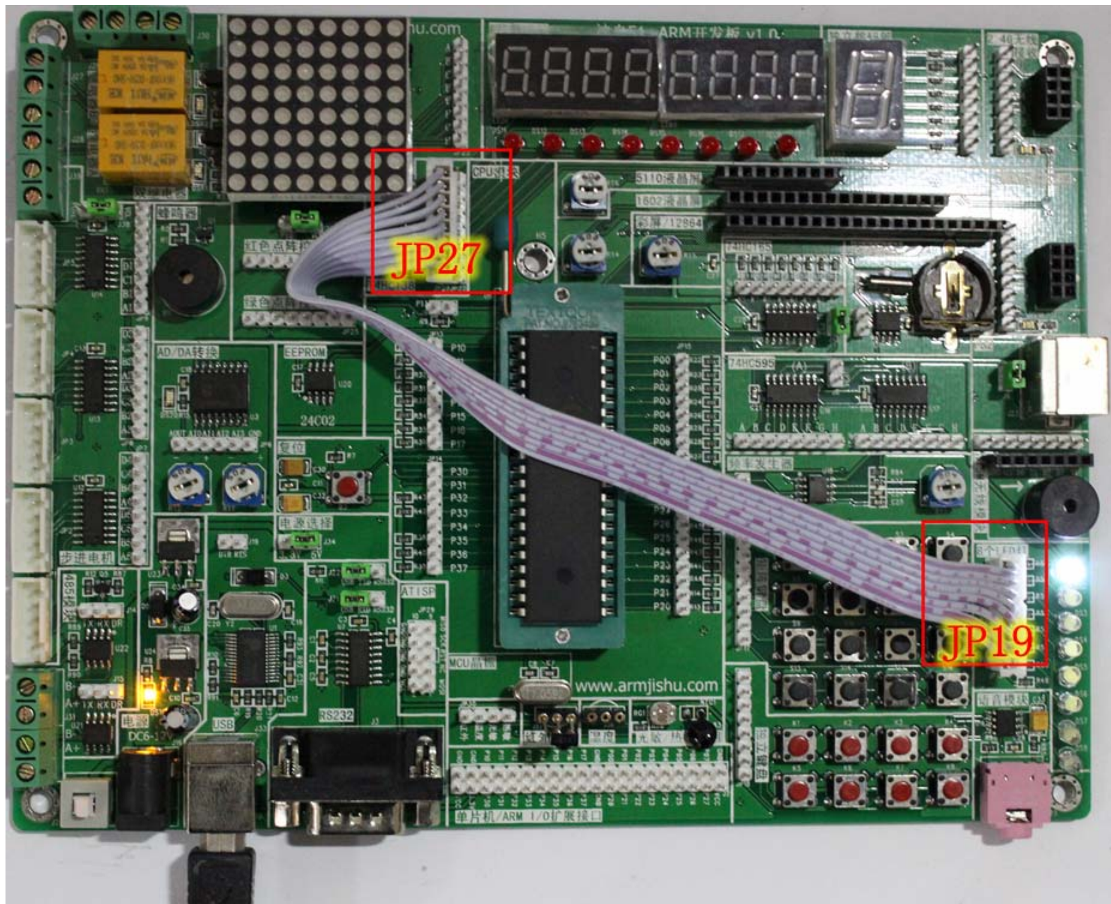


图 3-96 硬件连接实物图

我们看下程序

```
sbit HC138A = P2^2;    //定义译码器输入端 A 在 P2.2 管脚上
sbit HC138B = P2^3;    //定义译码器输入端 B 在 P2.3 管脚上
sbit HC138C = P2^4;    //定义译码器输入端 C 在 P2.4 管脚上
```

因为我们的神舟 51+ARM 单片机板子的译码器连接到了 51 单片机芯片上的 P22、P23 与 P24 引脚上，如下图 3-97。所以我们要先定义译码器的 3 个输入接口接到单片机的那 3 个 I/O 口。

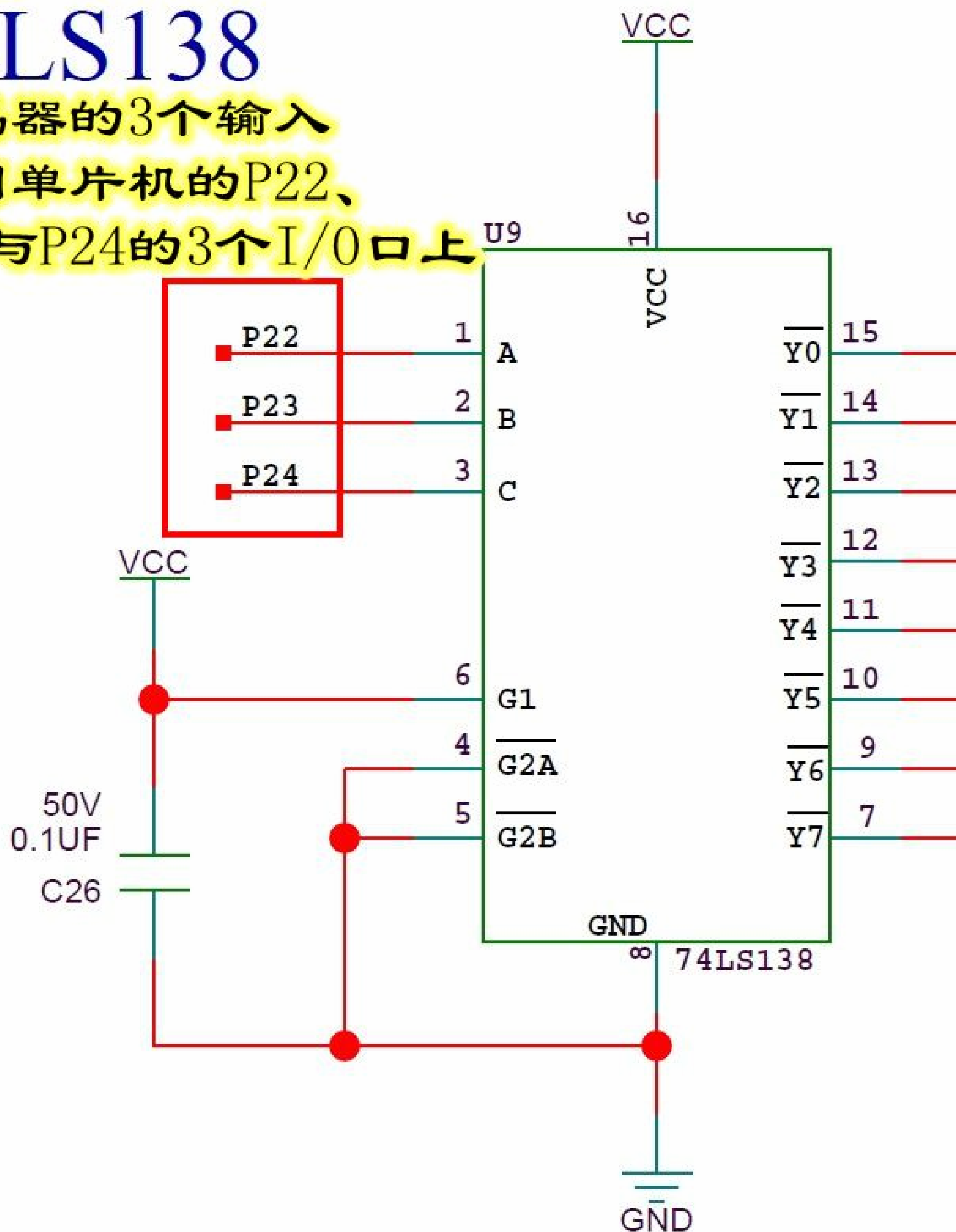


图 3-97 138 译码器原理图

我们再来看下下一条程序

```
void main (void)
{
    // 点亮第一个 LED 灯
    HC138C = 0; HC138B = 0; HC138A = 0;//译码器输入 000,输出为 1111 1110
}
```

这条程序是给译码器的 3 个输入输入“000”，根据上面的逻辑功能可以得出 Y7~Y0 输出 1111 1110，Y0 输出一个低电位，其他都为高，根据下面 51+ARM 单片机板子的电路图可以看到 8 个 LED 是共阳接法的，也就是说只需要单片机提供一个低电平就能点亮，而我们再看下译码器的输出 1111 1110 中只有一个输出 Y0 是低电平的，所以我们用排线连接译码器的输出到 LED 上就能看到点亮了 DS1 这个 LED 灯，如下图 3-98。

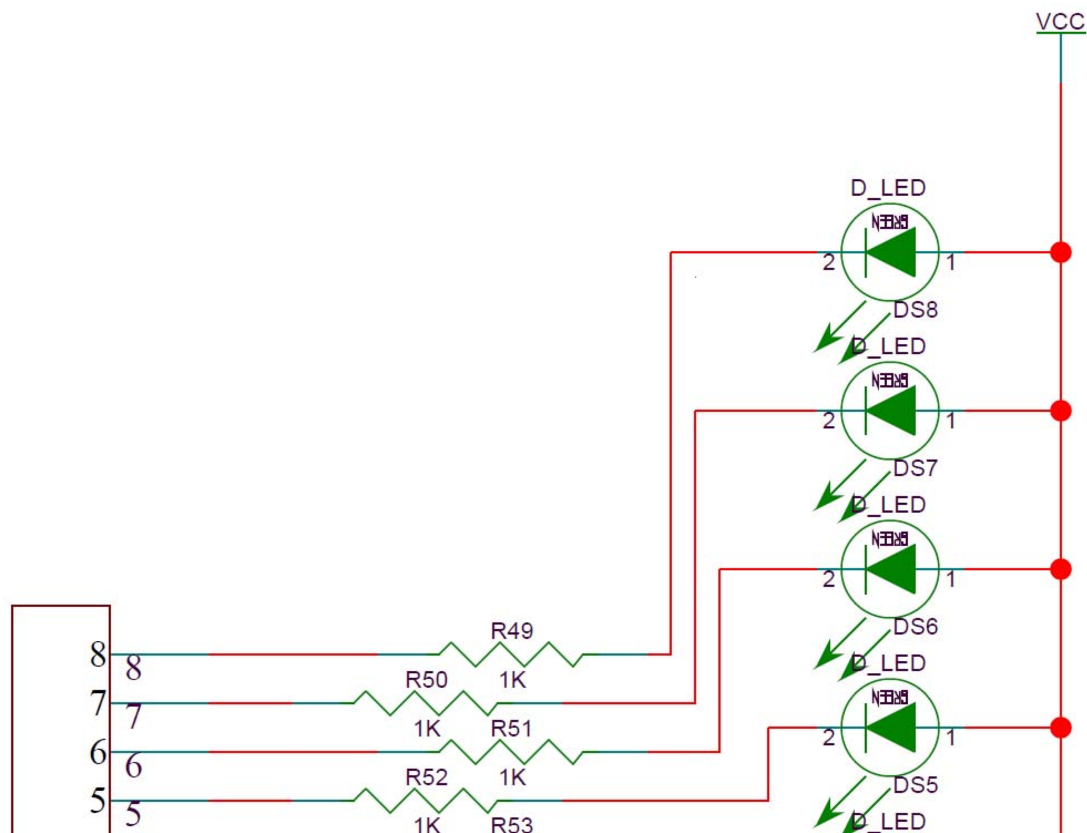


图 3-98 LED 部分原理图

知识要点：

我们通过单片机控制三八译码器输入，三八译码器输出来控制 1 个 LED 灯点亮。

HC138 译码器对应译码如下：

- 三位输入 000 对应八位输出为 1111 1110
- 三位输入 001 对应八位输出为 1111 1101
- 三位输入 010 对应八位输出为 1111 1011
- 三位输入 011 对应八位输出为 1111 0111
- 三位输入 100 对应八位输出为 1110 1111
- 三位输入 101 对应八位输出为 1101 1111
- 三位输入 110 对应八位输出为 1011 1111
- 三位输入 111 对应八位输出为 0111 1111

3.15.6 更多HC138 译码器例程

更多 HC138 译码器相应的例程可通过北京航空航天大学出版社下载专区下载，如下表 3-61：

表 3-61 译码器实验（HC138）更多丰富例程（**含详细注释和文档分析**）

序号	例程功能
例程 01	三八译码器点亮 1 个 LED 灯
例程 02	三八译码器循环点亮 LED 灯

3.16 锁存器（HC573）

3.16.1 什么是锁存器

锁存器(Latch)顾名思义，锁存，就是把信号暂存以维持某种电平状态。因为在某些领域，比如在 LED 和数码管显示方面，需要维持一个数据长时间不停的显示同样的内容，这样我们可以锁存一段信号线，持续的给予这个目标设备同样的不变化的信号，这样就可以节约 CPU 的资源，并且还能够单独实现持续的快速的刷新，如果这些工作都由 CPU 来完成，那么就会占用大量的 CPU 资源，所以单独出来一个锁存器芯片，来负责这样的工作，是节约资源的一种最佳的解决方案，当然也有助于 CPU 的寿命，因为如果长期工作锁存器如果坏了，而不会影响到 CPU。

比如在电子广告牌的显示设备上，大概每三十毫秒就要刷新一次，人眼就可以感觉到这个画面是可以接受的，是清晰的，不闪烁；如果完全用 CPU 来控制而不用锁存器，这就大大占用了处理器的处理时间，消耗了处理器的处理能力，还浪费了处理器的功耗。

并且 10 不同的广告牌各显示一个画面怎么控制了，实际上用 CPU 外加 10 个锁存器就可以实现，每次只更新一个画面到锁存器，另外 9 个锁存器持续锁存负责控制广告牌显示；一个一个轮流来，锁存器很便宜，这样就节约了 CPU 资源，降低了成本，而且锁存器的控制很简单，硬件复杂度也将有非常大的降低。

3.16.2 锁存器的实现原理

可以看下图 3-99，D7 是外部输入到锁存器里面的数据管脚，Q7 是锁存器输出管脚；当 LE=0 时，Q7 不受 D7 的影响；当 LE=1 时，Q7 随着 D7 的变化而变化，这里还有个非常关键的 OE（OUTPUT ENABLE）OUTPUT ENABLE 1 来控制输出端。

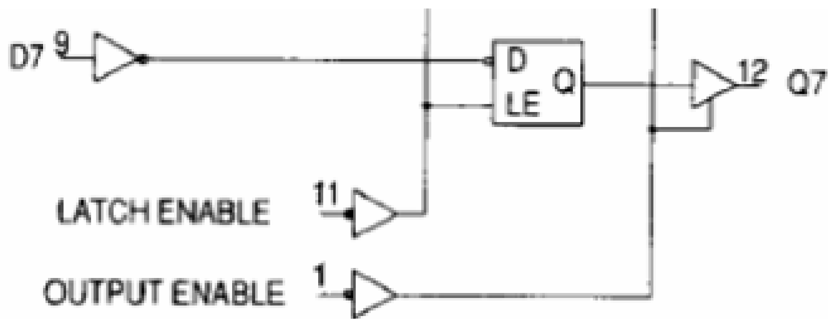


图 3-99 锁存器部分引脚结构图

总的来说，就是 LE 和 OE 是两个道关卡来控制锁存器，OE 为低电平，LE 为高电平的时候才会打开，那么输出 Q7 会随着 D7 的变化而变化；如果要锁存信号，OE 和 LE 都为低电平的时候，无论输入数据管脚 D7 如何变化，都不会影响到 Q7 的变化，达到锁存目的。真值表如下表 3-62 所示

表 3-62 74HC573 真值表

INPUTS 输入		Outputs 输出		
OE	LE	D	Q(HC573)	Q(HC563)
H	X	X	Z	Z
L	L	X	NO CHANGE	NO CHANGE
L	H	L	L	H
L	H	H	H	L

下图 3-100 是整个 74HC573 芯片的原理架构，可以看到 D0~D7 对应着输出 Q0~Q7，另外还有两个控制管脚 LE 和 OE。

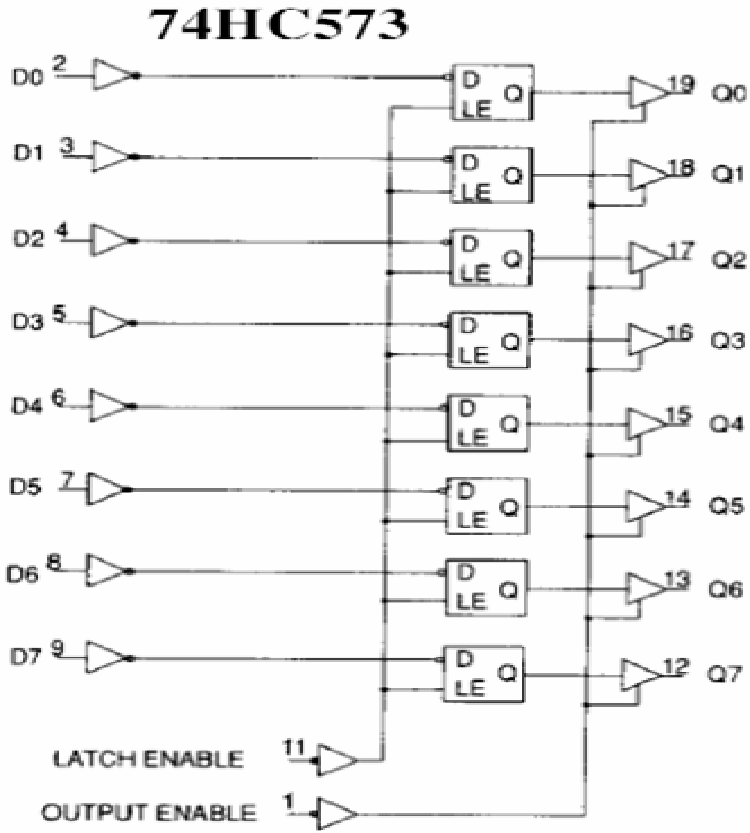


图 3-100 HC573 锁存器内部结构图

3.16.3 锁存器HC573 芯片介绍

下图 3-101 为我们单片机板子所用到的 HC573 的管脚功能图：

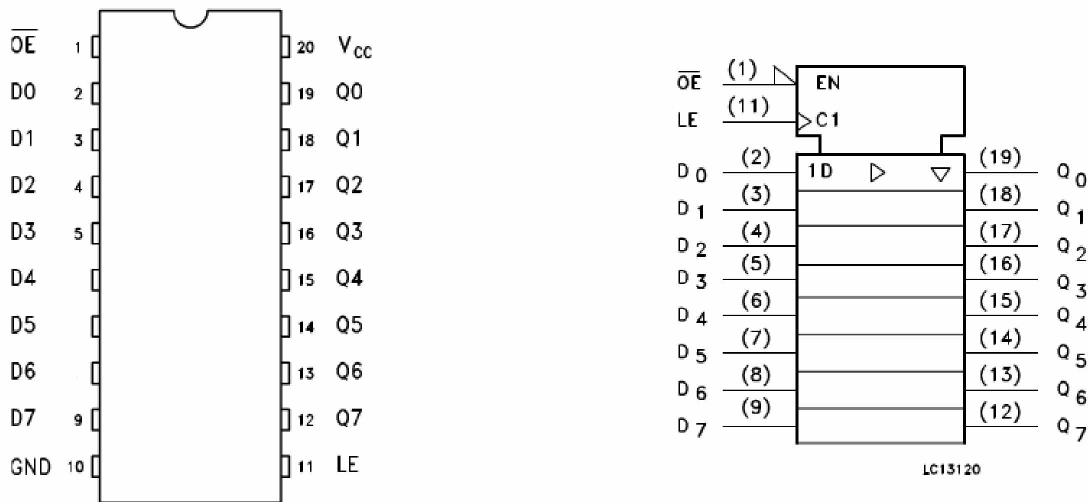


图 3-101 HC573 管脚图与 HC573 功能图

功能表如下表 3-63 所示：

表 3-63 HC573 引脚功能表

引脚号	符号	名称及功能
1	OE	3 态输出使能输入（低电平）
2、3、4、5、6、7、8、9	D0 to D7	数据输入
12、13、14、15、16、17、18、19	Q0 to Q7	3 态锁存输出
11	LE	锁存使能输入
10	GND	接地
20	VCC	电源电压

可以看到，其中 D0~D7 为数据输入端，Q0~Q7 为数据输出端，1 脚与 11 脚分别为输出使能引脚与锁存器使能引脚，剩下的 2 个 10 脚与 20 引脚分别为地与电源。我们接下来再看看如何把它设计到硬件原理图中。

3.16.4 硬件原理与连接

硬件连接原理图如图 3-102 所示：

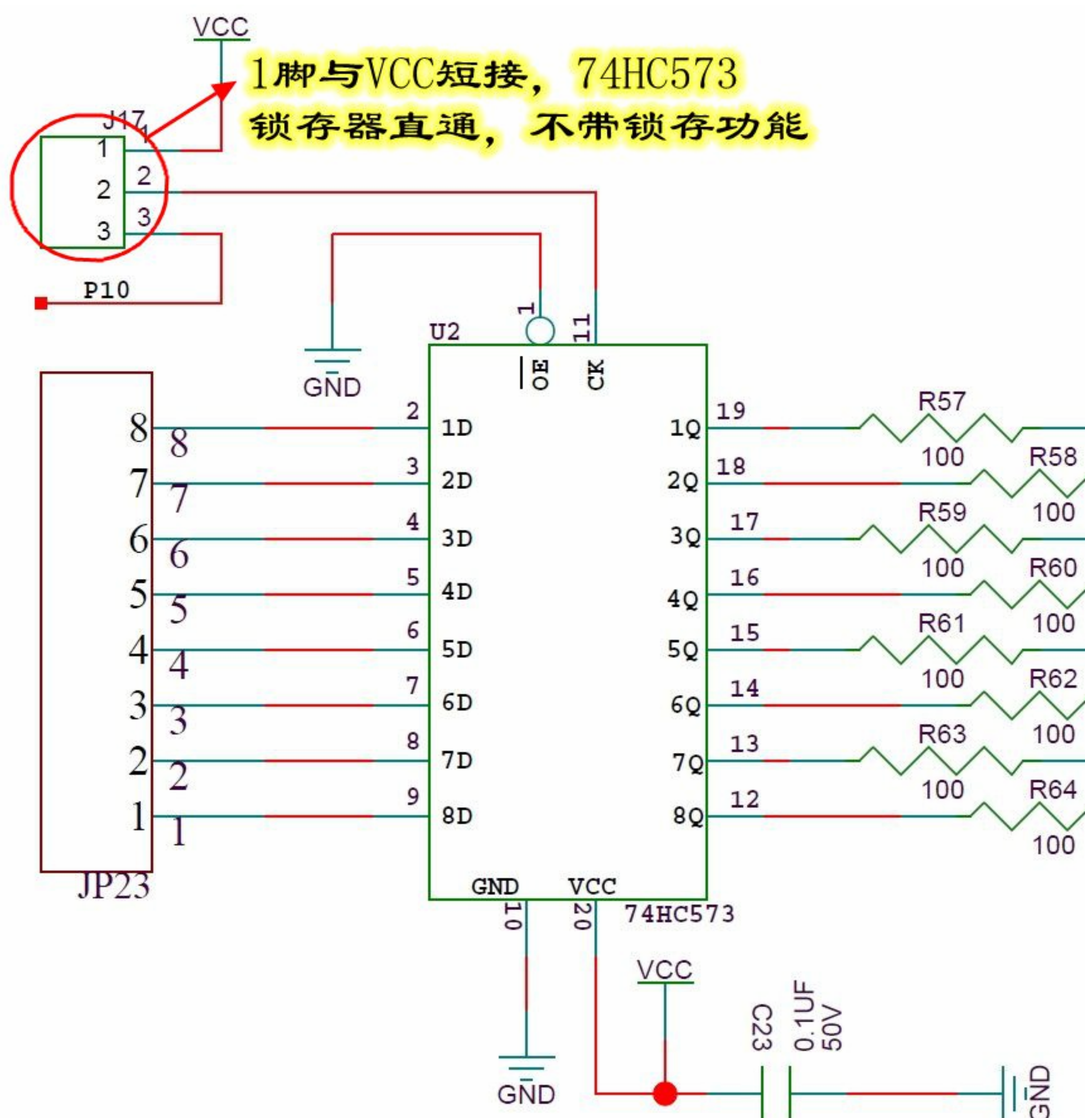


图 3-102　74HC573 硬件原理图

J17 用一个跳帽选择 LE 是连接 VCC 还是连接 P10 管脚来灵活控制，如果一直连 VCC 的话，只要 OE 为低(可以看到 OE 被连接到 GND，默认是强制拉低的)，这个锁存器就不具备锁存功能，是直通的，不具备锁存功能；除非 LE 由 P10 来控制，如果 LE 变成低电平，才可以完成锁存功能。

这里的 1D~8D 表示 8 个数据输入管脚，1Q~8Q 表示 8 个输出管脚，输出管脚每个管脚都挂了一颗限流电阻，以限制电流大小。真值表如下表 3-64 所示：

表 3-64 74HC573 真值表

INPUTS 输入		Outputs 输出		
OE	LE	D	Q(HC573)	Q(HC563)
H	X	X	Z	Z
L	L	X	NO CHANGE	NO CHANGE
L	H	L	L	H
L	H	H	H	L

3.16.5 例程 01 IO口高低电平控制点亮一个LED灯

代码如下：

```
/*****************************************************
* 例程：IO 口高低电平控制点亮一个 LED 灯
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：点亮 P0 口的一个 LED 灯
        该程序是单片机学习中最简单最基础的，
        通过程序了解如何控制端口的高低电平
*****************************************************/
#include<reg52.h>       //包含头文件，一般情况不需要改动，
                        //头文件包含特殊功能寄存器的定义
sbit LED_Ctrl=P1^1;     //用 sbit 关键字 定义 LED 到 P1.1 端口，
                        //LED 是自己任意定义且容易记忆的符号
/*--------------------------------------------
                主函数
--------------------------------------------*/
void main (void)
{
    LED_Ctrl= 0;   //初始化 P1^1 端口，只有为低电平时，才可点亮动态 LED 灯。
 /* 使用 1 个字节(即 8 个 bit 位)对单个端口赋值（P0 端口是由 P0.0~P0.7 总共 8 个位组
成）*/
      P0 = 0x00;          //P0 口全部为低电平，对应的 LED 灯全灭掉
                          //16 进制 0x00 换算成二进制是 0000 0000

      P0 = 0x01;           //P0 口的最低位点亮，可以更改数值是其他的灯点亮
                           //0x01 是 16 进制，0x 开头表示 16 进制数，
                           //01 换算成二进制是 0000 0001
}
```

硬件连接对应关系如表 3-65 所示：

表 3-65 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P0	JP15（A 向左）	直连	JP23（A 向右）	8 位排线	控制单个 LED 灯
J17 短接 VCC 端					
短接 J9（连接 P1.1 管脚）					
实验现象：下载程序后，我们可以看到 1 个 LED 灯点亮					

知识要点：

LED_Ctrl= 0; //初始化 P1^1 端口，只有为低电平时，才可点亮动态 LED 灯。

上面的程序中，我们先为 P1.1 提供一个低电平。而 LED 的阴极都接到该引脚上，那样的话，只需要为 LED 的阳极提供一个高电平就能点亮。

P0 = 0x00;   //P0 口全部为低电平，对应的 LED 灯全灭掉

//16 进制 0x00 换算成二进制是 0000 0000

再下来就是 P0 口输出全为低电平，把与 P0 口连接的 LED 都熄灭了，防止影响到最终的结果，有些 LED 可能接收到 P0 口本身输出的高电平而导致点亮该 LED 的，所以我们这里需要为 P0 口输出低电平，熄灭全部的 LED 灯

```
P0 = 0x01;          //P0 口的最低位点亮，可以更改数值是其他的灯点亮
                    //0x01 是 16 进制，0x 开头表示 16 进制数，
                    //01 换算成二进制是 0000 0001
```

P0 端口中的 1 位 I/O 口输出高电平，点亮该路上的 LED；

知识要点：

1. 当 74HC573 锁存器 LE =1 时，P0 端口的 8 位数据线与 74HC573 内部数据保持器的输入端连通。

2. 这 8 个 LED 灯与之前的 LED 灯不是同一组。

3. P1.1 连接了所有 LED 灯端的阴极，程序中将 P1.1 置成低电平，然后用 P0 的 8 个管脚对 LED 灯的阳极输出高电平，就可以点亮 LED 灯。原理图如下图 3-103 所示：

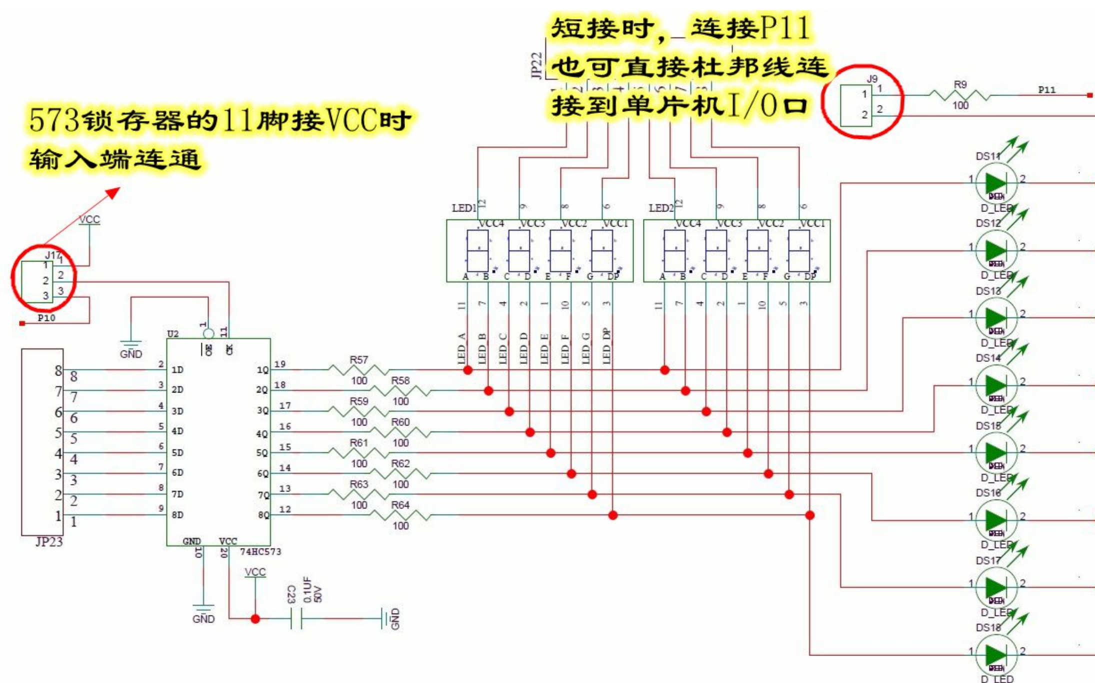


图 3-103 锁存器控制 LED 功能原理图

3.16.6 更多有关HC573 锁存器例程

更多 HC573 锁存器相应的例程可通过北京航空航天大学出版社下载专区下载，如下表 3-66：

表 3-66 锁存器（HC573）更多丰富例程（含详细注释和文档分析）

序号	例程功能
例程 01	IO 口高低电平控制点亮一个 LED 灯
例程 02	IO 输出-点亮多个 LED 灯
例程 03	IO 输出闪烁 1 个 LED 灯

例程 04	IO 输出控制 8 位 LED 右移
例程 05	IO 输出闪烁 1 个 LED 灯（带锁存功能）
例程 06	IO 输出控制 8 位 LED 右移（带锁存功能）

3.17 PS2 键盘输入

3.17.1 PS/2 接口简介

PS/2 原是“Personal System 2”的意思，“个人系统 2”，是 IBM 公司在上个世纪 80 年代推出的一种个人电脑。以前完全开放的 PC 标准让 IBM 觉得利益受了损失。所以 IBM 设计了 PS/2 这种电脑，目的是重新定义 PC 标准，不再采用开放标准的方式。在这种电脑上 IBM 使用了新型 MCA 总线，新的 OS/2 操作系统。PS/2 电脑上使用的键盘鼠标接口就是现在的 PS/2 接口。因为标准不开放，PS/2 电脑在市场中失败了。只有 PS/2 接口一直沿用到今天。

本章节通过 51 单片机的 IO 模拟 PS/2 协议，接收来自 PS/2 键盘的按键数据，模拟之前首先看下 PS/2 的硬件接口，再看下 PS/2 的协议，然后再用 51 单片机的 IO 接口模拟：

3.17.2 PS/2 键盘鼠标的硬件接口

PS/2 接口是在较早电脑上常见的接口之一，用于鼠标、键盘等设备。一般情况下，PS/2 接口的鼠标为绿色，键盘为紫色，PS/2 接口外观如下图 3-104、3-105 所示：



图 3-104 PS/2 接口实物图图



3-105 PS/2 接口实物图

PS/2 接口是一种 6 针的圆型接口。但只使用其中的 4 针传输数据和供电，其余 2 个为空脚，其中“Male”表示公的、插头;“Female”母的，插座管脚图如下图 3-106：

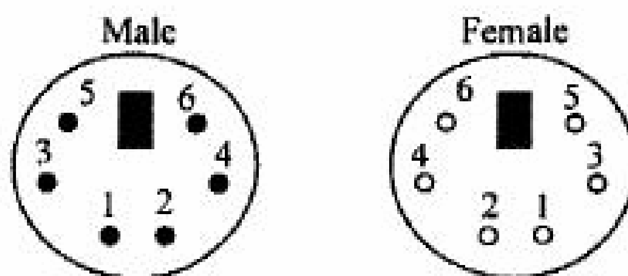


图 3-106　PS/2 接口管脚图

管脚定义如下表 3-67：

表 3-67 PS/2 管脚定义

管脚	信号名	描述
1	DATA	按键数据(Key Data)
2	n/c	空(Not connected)
3	GND	地(Gnd)
4	VCC	5V电源(Power, +5 VDC)
5	CLK	时钟(Clock)
6	n/c	空(Not connected)

3.17.3 PS/2 的协议

可以看到 PS/2 实际最少可以由两根管脚进行控制，一根是 CLK 时钟信号线，一根是数据线，按照一定的规则，把数据传输到另外一端就可以了。

PS/2 每笔数据传输所的时序图如下：

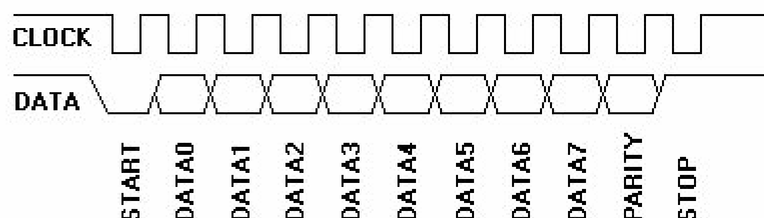


图 3-107　数据传输时序图

具体解释这个时序图的协议过程如下表 3-68 所示：

表 3-68 协议过程

每帧包含 11 位，每位在时钟的下降沿被主机读	
1 个起始位	总是逻辑 0
8 个数据位	低位（LSB）在前
1 个奇偶校验位	奇校验
1 个停止位	总是逻辑 1

PS/2 通信协议是一种双向同步串行通迅协议。通迅的两端通过 CLOCK（时钟信号端）同步，并通过 DATA（数据端口）交换数据。任何一方如果想要抑制另外一方的通迅时，只

需要把 CLOCK 拉到低电平。

PS2 标准，规范每笔数据传输包含起始位(start bit)、扫描码(scan code)、奇同位检查(odd parity)、以及终止位(stop bit)共计 11 位，并以双向串行数据传输的方式，达到通信的目的。且当主机端(host)或从机端(slave)并无传送或接收数据时，数据传输端口及时钟信号端口均将升为高电位。

在这里明白了协议沟通的方式，数据就是这么进行交换的，下面用一个实际例子来看一下具体怎么使用 PS/2 协议的。

3.17.4 键盘与PS/2 协议实例分析

键盘其实就是一个大型的按键矩阵，它们由安装在电路板上的处理器（叫做“键盘编码器”）来监视着。虽然不同的键盘可能采用不同的处理器，但是它们完成的任务都是一样的，即监视哪些按键被按下，哪些按键被释放了，并将这些信息传送到主机。如果有必要，处理器处理所有的去抖动，并在它的 16 字节的缓冲区里缓冲数据。主机端包含了一个“键盘控制器”与键盘处理器进行通讯，并解码来自键盘处理器的信息，然后高速系统当前按键对应的处理事情。主机与键盘之间的通讯仍旧采用 IBM 的协议。

键盘处理器花费很多时间来扫描或监视按键矩阵。如果发现有按键按下、释放或长按，键盘就发送“扫描码”的信息到主机。扫描码有两种不同的类型：“通码”和“断 码”。当一个键被按下去或长按的时候，键盘就发送通码；当一个键被释放的时候，键盘就发送断码。每个键盘被分配了唯一的通码和断码，这样主机通过查找唯一的扫描码就可以确定是哪个按键被按下或释放。

每个键一整套的通断码组成了“扫描码集”，现在大多数的键盘采用第三套扫描码。由于没有一个简单的公式可以计算扫描码，所以要知道某个特定按键的通码和断码，只能采用查表的方法来获得。需要特别注意的是，按键的通码值表示键盘上的一个按键，并不表示印刷在按键上的那个字符，这就意味着通码和 ASCII 码之间没有任何关联。

在与键盘通信的时候，PS/2 接口是输入接口，而不是传输接口。所以 PS2 口根本没有传输速率的概念，只有扫描速率。在 Windows 环境下，PS/2 鼠标的采样率默认为 60 次/秒，USB 鼠标的采样率为 120 次/秒。较高的采样率理论上可以提高鼠标的移动精度。PS/2 接口设备不支持热插拔，强行带电插拔有可能烧毁主板。

扫描到键盘数据之后，传输给中心系统进行解码分析，然后进行相应的操作。

第三套键盘扫描码可以参考一下，比如字符’A’按下是发送数据 1C，弹起按键发送数据 F0,1C，具体其他整张表的数据如下表 3-69：

表 3-69　键盘扫描码按下与弹起数据的发送

按键	按下	弹起	按键	按下	弹起	按键	按下	弹起
A	1C	F0,1C	9	46	F0,46	[	54	F0,54
B	32	F0,32	`	0E	F0,0E	INSERT	67	F0,67
C	21	F0,21	-	4E	F0,4E	HOME	6E	F0,6E
D	23	F0,23	=	55	F0,55	PG UP	6F	F0,6F
E	24	F0,24	\	5C	F0,5C	DELETE	64	F0,64
F	2B	F0,2B	BKSP	66	F0,66	END	65	F0,65
G	34	F0,34	SPACE	29	F0,29	PG DN	6D	F0,6D

H	33	F0,33	TAB	0D	F0,0D	U ARROW	63	F0,63
I	43	F0,48	CAPS	14	F0,14	L ARROW	61	F0,61
J	3B	F0,3B	L SHFT	12	F0,12	D ARROW	60	F0,60
K	42	F0,42	L CTRL	11	F0,11	R ARROW	6A	F0,6A
L	4B	F0,4B	L WIN	8B	F0,8B	NUM	76	F0,76
M	3A	F0,3A	L ALT	19	F0,19	KP /	4A	F0,4A
N	31	F0,31	R SHFT	59	F0,59	KP *	7E	F0,7E
O	44	F0,44	R CTRL	58	F0,58	KP -	4E	F0,4E
P	4D	F0,4D	R WIN	8C	F0,8C	KP +	7C	F0,7C
Q	15	F0,15	R ALT	39	F0,39	KP EN	79	F0,79
R	2D	F0,2D	APPS	8D	F0,8D	KP .	71	F0,71
S	1B	F0,1B	ENTER	5A	F0,5A	KP 0	70	F0,70
T	2C	F0,2C	ESC	08	F0,08	KP 1	69	F0,69
U	3C	F0,3C	F1	07	F0,07	KP 2	72	F0,72
V	2A	F0,2A	F2	0F	F0,0F	KP 3	7A	F0,7A
W	1D	F0,1D	F3	17	F0,17	KP 4	6B	F0,6B
X	22	F0,22	F4	1F	F0,1F	KP 5	73	F0,73
Y	35	F0,35	F5	27	F0,27	KP 6	74	F0,74
Z	1A	F0,1A	F6	2F	F0,2F	KP 7	6C	F0,6C
0	45	F0,45	F7	37	F0,37	KP 8	75	F0,75
1	16	F0,16	F8	3F	F0,3F	KP 9	7D	F0,7D
2	1E	F0,1E	F9	47	F0,47	]	5B	F0,5B
3	26	F0,26	F10	4F	F0,4F	;	4C	F0,4C
4	25	F0,25	F11	56	F0,56	’	52	F0,52
5	2E	F0,2E	F12	5E	F0,5E	,	41	F0,41
6	36	F0,36	PRNT SCRN	57	F0,57	.	49	F0,49
7	3D	F0,3D	SCROLL	5F	F0,5F	/	4A	F0,4A
8	3E	F0,3E	PAUSE	62	F0,62			

3.17.5 单片机与PS/2 设备连接的硬件原理图

PS/2 硬件原理图如下图 3-108：

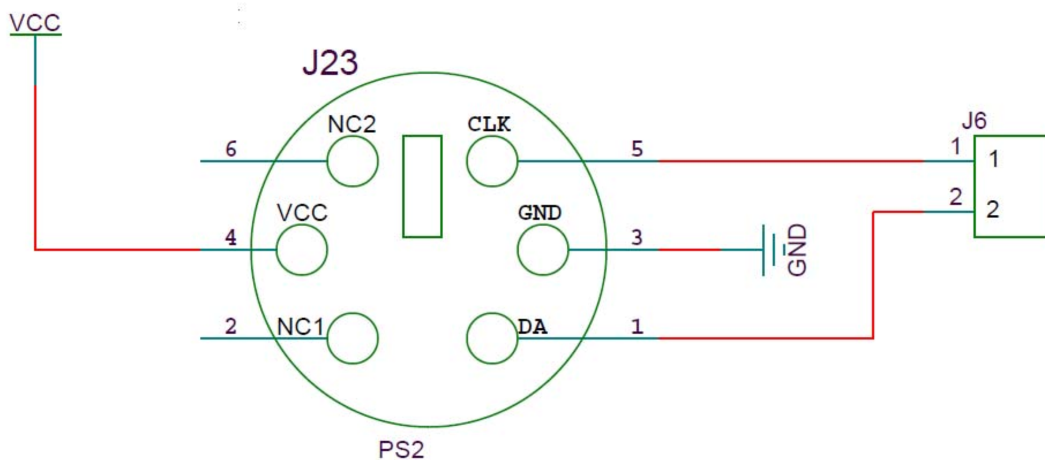


图 3-108 PS/2 硬件原理图

可以看到原理图中的 CLK 是 PS/2 协议中的时钟线，DA 是 PS/2 协议中的数据数据线；在这里通过单片机对这两根线进行连接完成对整个 PS/2 接口的模拟和控制。

3.17.5 例程 01 PS2 键盘输入在LED数码管显示

代码如下：

```
/*************************************************************************
*  例程：接收标准PS2键盘输入在数码管上显示
*  作者：www.armjishu.com
*  内容：接收标准PS2键盘输入，在数码管上显示
*  现象：此程序使用标准PS2键盘输入，在数码管上动态显示
*          本键盘使用数字测试，其他按键不能使用，用户可以自行扩展。
*          由于开发板和程序的各种参数，程序中没有使用奇偶校验，
*          不保证没有误码,校验程序请自行添加。
*************************************************************************/
/*  包含头文件，一般情况不需要改动，头文件包含特殊功能寄存器的定义  */
#include <reg52.h>
#include "display.h"
sbit Key_Data = P3^3;              //定义Keyboard引脚
sbit Key_CLK    = P3^2;            //使用中断
unsigned char ScanValue;
unsigned char KeyValue;
unsigned char IntNum=0;
bit Flag=0;
extern unsigned char LedData[8];
extern unsigned char code DuanMa[];
/*------------------------------------------------------------
                        函数声明
------------------------------------------------------------*/
```

```
void PS2_Init(void);
void Decode(unsigned char ScanCode);
/*------------------------------------------------------------
                          主函数
------------------------------------------------------------*/
void main (void)
{
    unsigned char LedFlag = 0;
    PS2_Init();          //初始化PS/2接口
    Init_Timer0();       //定时器初始化，由于数码管动态显示
    Flag = 0;
    while (1)
    {
        if (Flag)
        {
            // 当收到0xF0，Key_UP置1表示断码开始
            if (ScanValue != 0xF0)
            {
                Decode(ScanValue);      //解码
                Flag = 0;               //标识字符处理完了
                //如果串口得到的数据为数字(0-9)，则在数码管上显示
                if((KeyValue >= 0) && (KeyValue <= 0xF))
                {
                    LedData[LedFlag] = DuanMa[KeyValue];
                    LedFlag++;
                    if(LedFlag > 7)
                    {
                        LedFlag = 0;
                    }
                    LedData[LedFlag] = 0x08;      //'_'
                }
            }
        }
        else
        {
            EA = 1;        //开中断
        }
    }
}
/*---------------------------------------------------
                    外部中断读入信息
键盘和鼠标使用一种每帧包含11位的串行协议如下
1个起始位           总是逻辑0
8个数据位           低位（LSB）在前
```

```
1个奇偶校验位    奇校验
1个停止位        总是逻辑1
----------------------------------------------------*/
void Keyboard_out(void) interrupt 0
{
    //IntNum等于0时为起始位，1到8为8个数据位
    if ((IntNum > 0) && (IntNum < 9))
    {
        //因键盘数据是低>>高，结合上一句所以右移一位
        ScanValue = ScanValue >> 1;
        if (Key_Data)
        //当键盘数据线为1时暂存到最高位
        ScanValue = ScanValue | 0x80;
    }
    IntNum++;
    while (!Key_CLK);          //等待PS/2CLK拉高
    if (IntNum > 10)
    {
        //中断11次后表示一帧数据收完，清变量准备下一次接收
        IntNum = 0;
        Flag = 1;                //标识有字符输入完了
        //EA = 0;                //关中断等显示完后再开中断
    }
}
/*----------------------------------------------------
                    解码信息
本函数只解析键盘第二行的数字键和数字小键盘的数字及ABCDEF
----------------------------------------------------*/
void Decode(unsigned char ScanCode)
{
    switch (ScanCode)
    {
        case 0x45 : //第二行的数字
        case 0x70 : //数字小键盘的数字
            KeyValue = 0;
            break;
        case 0x16 :
        case 0x69 :
            KeyValue = 1;
            break;
        case 0x1E :
        case 0x72 :
            KeyValue = 2;
            break;
```

```
case 0x26 :
case 0x7A :
        KeyValue = 3;
        break;
case 0x25 :
case 0x6B :
        KeyValue = 4;
        break;
case 0x2E :
case 0x73 :
        KeyValue = 5;
        break;
case 0x36 :
case 0x74 :
        KeyValue = 6;
        break;
case 0x3D :
case 0x6c :
        KeyValue = 7;
        break;
case 0x3E :
case 0x75 :
        KeyValue = 8;
        break;
case 0x46 :
case 0x7d :
        KeyValue = 9;
        break;
case 0x1C :
        KeyValue = 0xA;
        break;
case 0x32 :
        KeyValue = 0xB;
        break;
case 0x21 :
        KeyValue = 0xC;
        break;
case 0x23 :
        KeyValue = 0xD;
        break;
case 0x24 :
        KeyValue = 0xE;
        break;
case 0x2B :
```

```
				KeyValue = 0xF;
				break;
			default :
				KeyValue = 0xFF;
				break;
		}
}
/*----------------------------------------------------
	ps2初始化（实际初始化外部中断）
----------------------------------------------------*/
void PS2_Init(void)
{
	IT1 = 0;			//设外部中断1为低电平触发
	EA   = 1;			//外部中断开
	IP   = 1;			//外部中断0的优先级设为最高
	EX0 = 1;			//开中断
}
```

本实验神舟51开发板的线缆配置如下表3-70所示

表3-70 硬件连接配置

单片机接口	插座 1	方式	插座 2	线缆	功能
P3^3	JP14	直连	J6	1 根杜邦线	PS/2 接口的 DATA
P3^2	JP14	直连	J6	1 根杜邦线	PS/2 接口的 CLK
P0	JP15(A 向左)	直连	JP23(B 向左)	8 芯排线	数码管数据位
P22-P24	JP27	跳帽	JP22	8 个跳帽	74LS138 数码管位选
实验现象：如键盘上的数字按键和 ABCDEF 可以在数码管上显示则说明操作成功，否则操作失败。					

连接图如下图3-109所示：

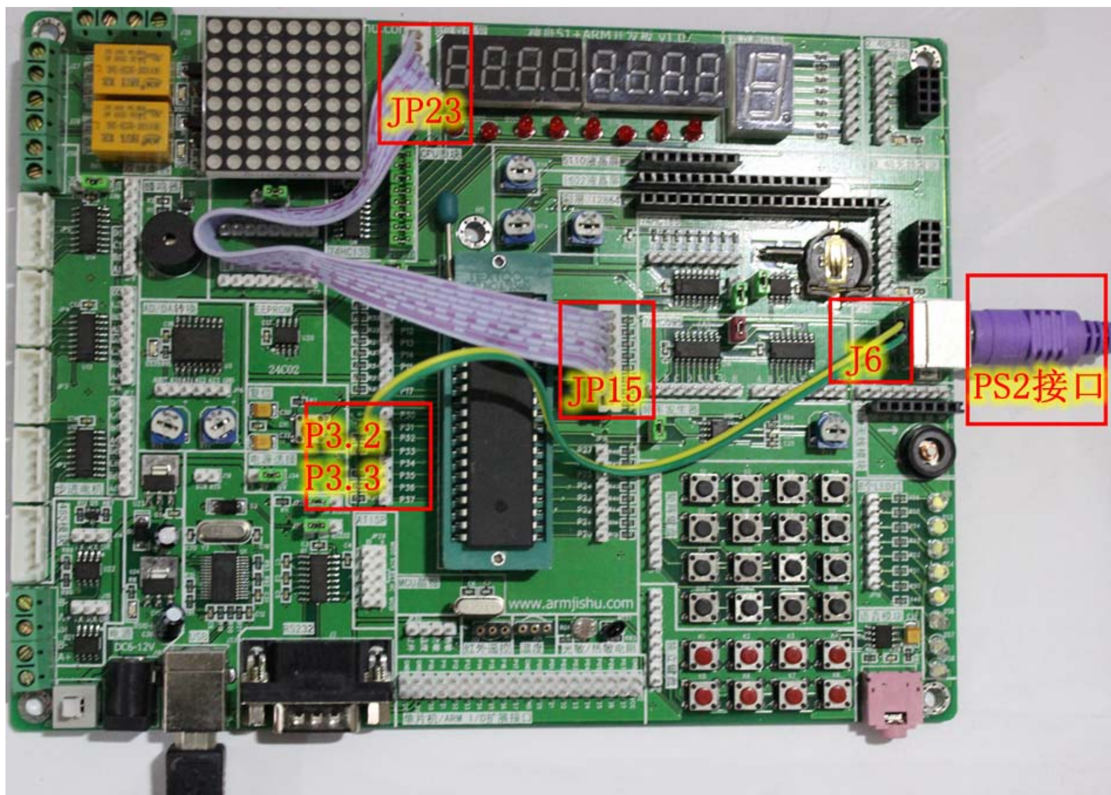


图 3-109 硬件连接实物图

本实验是通过 51 单片机的 IO 模拟 PS/2 协议，接收来自 PS/2 键盘的按键数据，在 LED 数码管上显示。通过本例程了解 PS/2 接口的基本原理，理解并掌握单片机接收 PS/2 键盘的按键数据的知识。

此程序使用标准 PS2 键盘输入，在数码管上动态显示本键盘使用数字测试，其他按键不能使用，用户可以自行扩展。由于开发板和程序的各种参数，程序中没有使用奇偶校验，不保证没有误码。键盘上的数字按键和 ABCDEF 可以在数码管上显示则说明操作成功，否则操作失败。

知识要点：

1.首先看第一个函数 PS2_Init（）函数的初始化，实际初始化外部中断

2. 函数 Init_Timer0（）定时器初始化，到时候会产生中断，来用于数码管动态显示，与 PS/2 协议无关。

```
/*------------------------------------------------
                定时器中断子程序
------------------------------------------------*/
void Timer0_isr(void) interrupt 1
{
    TH0=(65536-2000)/256;   //重新赋值 2ms
    TL0=(65536-2000)%256;
    Display(0,8);
}
```

这里是定时器 0 的中断函数，定时器产生中断后，就会执行 Display()函数来刷新数码管的值，重复这样循环，使得数码管在人的视觉中产生一直显示的假象。

3.可以看到代码中将 P3.3 绑定到了 PS/2 的数据线上，将 P3.2 绑定到了 PS/2 的时钟线

上，下面图 3-110 所示可以通过操作 P3.3 和 P3.2 这两个管脚来模拟 PS/2 协议就可以了。

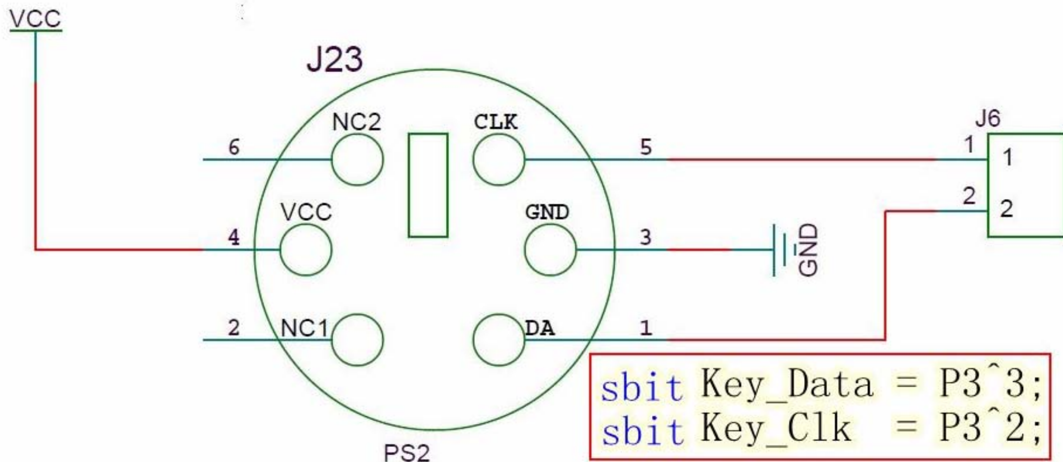


图 3-110　PS/2 数据接口

4.可以看到，P3.2 是外部中断线，每一个 PS/2 的 CLK 时钟信号一来，都会产生一个中断，产生中断就会执行这个 keyboard_out()中断函数，这个函数从 PS/2 协议中可以得知，应该要完成读取数据的功能，一个时钟周期就要读取一个数据 bit，下节来具体分析这个函数。

```
/*----------------------------------------------------------
                外部中断读入信息
键盘和鼠标使用一种每帧包含 11 位的串行协议如下
1 个起始位          总是逻辑 0
8 个数据位          低位（LSB）在前
1 个奇偶校验位      奇校验
1 个停止位          总是逻辑 1
----------------------------------------------------------*/
void Keyboard_out(void) interrupt 0
{
    //IntNum 等于 0 时为起始位，1 到 8 为 8 个数据位
    if ((IntNum > 0) && (IntNum < 9))
    {
        //因键盘数据是低>>高，结合上一句所以右移一位
        ScanValue = ScanValue >> 1;
        if (Key_Data)
        //当键盘数据线为 1 时暂存到最高位
        ScanValue = ScanValue | 0x80;
    }
    IntNum++;
    while (!Key_CLK);          //等待 PS/2CLK 拉高
    if (IntNum > 10)
    {
        //中断 11 次后表示一帧数据收完，清变量准备下一次接收
        IntNum = 0;
        Flag = 1;              //标识有字符输入完了
```

```
            //EA = 0;                  //关中断等显示完后再开中断
        }
    }
```

5. 分析 keyboard_out()函数，看下数据如何被采集的

第一步：因为 key_data 是数据线，那么如果 key_data 为 1，那么将这个 1 通过与或操作赋值给 scanValue 变量中:

```
    sbit Key_Data = P3^3;          //定义 Keyboard 引脚
    if (Key_Data)
    //当键盘数据线为 1 时暂存到最高位
    ScanValue = ScanValue | 0x80;
```

第二步：每中断进来之后，都通过 IntNum++;来计数，并且通过‘if ((IntNum > 0) && (IntNum < 9))’来判断只采集 8 位 bit 给 ScanValue

第三步：每采集 1 个 bit，都会通过 while (!Key_CLK);等待时钟跳高，表示这 1 个时钟周期结束，可以传入第 2 个数据位 bit 了。

第四步：判断 if (IntNum > 10)表示中断 11 次后表示一帧数据收完，清变量准备下一次接收，对 PS/2 协议来说，它需要接收到的数据是一种每帧包含 11 位的串行数据：

1 个起始位　　总是逻辑 0
8 个数据位　　低位（LSB）在前
1 个奇偶校验位　奇校验
1 个停止位　　总是逻辑 1

在我们的代码中截取的是中间那 8 个数据位。

6.进入到 main()函数的 while 循环中进行解码，这个代码里目前只摘抄了 0-9 这 10 个数字的对应 PS/2 编码

```
    while (1)
    {
        if (Flag)
        {
            //  当收到 0xF0，Key_UP 置 1 表示断码开始
            if (ScanValue != 0xF0)
            {
                Decode(ScanValue);      //解码
                Flag = 0; //标识字符处理完了
                //如果串口得到的数据为数字(0-9)，则在数码管上显示
                if((KeyValue >= 0) && (KeyValue <= 0xF))
                {
                    LedData[LedFlag] = DuanMa[KeyValue];
                    LedFlag++;
                    if(LedFlag > 7)
                    {
                        LedFlag = 0;
                    }
                    LedData[LedFlag] = 0x08;       //'_'
```

```
                }
            }
        }
        else
        {
            EA = 1;         //开中断
        }
    }
```

7. 下面进入到 Decode()函数中，看看原来里面的 case 语句的数字已经跟键盘的 0~9 的字符都对应了 PS/2 编码的。如下图 3-111 所示

```
void Decode(unsigned char ScanCode)
{
    switch (ScanCode)
    {
        case 0x45 : //第二行的数字
        case 0x70 : //数字小键盘的数字
            KeyValue = 0;
            break;

        case 0x16 :
        case 0x69 :
            KeyValue = 1;
            break;

        case 0x1E :
        case 0x72 :
            KeyValue = 2;
            break;

        case 0x26 :
        case 0x7A :
            KeyValue = 3;
            break;

        case 0x25 :
        case 0x6B :
            KeyValue = 4;
            break;
```

按键	按下
1	16
2	1E
3	26
4	25
5	2E
6	36
7	3D
8	3E

图 3-111 字符对应 PS/2 编码

8. 识别到具体的按键之后，就进行相对应的操作，这里就不详细讲解了，具体细节请见代码。

3.17.6 更多PS/2 的例程以及分析

更多 PS/2 相应的例程可通过北京航空航天大学出版社下载专区下载，如下表 3-71：

表 3-71 PS2 的更多丰富例程

序号	例程功能
例程 01	PS2 键盘输入在 LED 数码管显示
例程 02	PS2 键盘输入从串口输出

3.18　A/D和D/A（PCF8591）

3.18.1 名词解释

1.模拟信号：模拟信号是指信息参数在给定范围内表现为连续的信号。 或在一段连续的时间间隔内，其代表信息的特征量可以在任意瞬间呈现为任意数值的信号。像那些电压/电流与声音这些都是模拟信号。

2.数字信号：数字信号指幅度的取值是离散的，幅值表示被限制在有限个数值之内。二进制码就是一种数字信号。二进制码受噪声的影响小，易于有数字电路进行处理，所以得到了广泛的应用。

3.A/D:从字面上看,A我们称为模拟信号(Analog signal),D我们称为数字信号(digital signal)，A/D 转换器也就是把模拟信号转换成数字信号的器件

4.D/A：D/A 转换器刚好与 A/D 功能相反，只是功能是反过来的，它是把数字信号转换为模拟信号的

3.18.2 模拟转数字信号和数字转模拟信号产生的背景

日常生活中，有很多事物是我们不能直观的表达出来的，像耳朵听到的声音大小，到数字电路中到底是多大的一个值？速度的快慢，可以感觉一个人跑得很快，也可以感觉一个人跑得很慢，那么具体是什么速度呢？这些例子还有很多，如何给一个确定的值呢？能不能把它们的这个确定的值给测量出来呢？答案是可以，这就是把一个感知的值，在这里暂且称之为模拟值转换成数字值，转换成单片机可以认识的数字的值，这样的转换在本章节叫做 A/D 模数转换；在仪器检测系统中，常常需要将检测到的连续变化的模拟量如：温度、压力、流量、速度等转换成离散的数字量，才能进行计算处理；这些模拟量通过传感器转换为电信号后，就需要通过一定的处理变成数字量，实现模拟量到数字量转换的设备，通常称为 ADC，也叫 A/D。

那什么是 D/A 转换呢？实际就是 A/D 反过来，叫做数模转换，比如将一个数字的 MP3 音乐转换成模拟型号，转换成可以在音响中或者耳机里可以听到的声音，这就是数字模拟转换的一个实际例子，也叫做 D/A 转换；

转换器种类繁多，DA 转换器的有：电压输出型的、电流输出型、乘算型等；AD 转换器的有：积分型、逐次比较型、并行比较型/串并行比较型等，它们的作用功能等我们就不一一介绍了，读者可以搜下资料就行了。

3.18.3 模数A/D的转换原理

逐次逼近转换过程和用天平称物重非常相似。天平称重物过程是，从最重的砝码开始试放，与被称物体行进比较，若物体重于砝码，则该砝码保留，否则移去。再加上第二个次重砝码，由物体的重量是否大于砝码的重量决定第二个砝码是留下还是移去。照此一直加到最小一个砝码为止。将所有留下的砝码重量相加，就得此物体的重量。仿照这一思路，逐次比较型 A/D 转换器，就是将输入模拟信号与不同的参考电压作多次比较，使转换所得的数字量在数值上逐次逼近输入模拟量对应值。

逐次逼近型 ADC 因其功耗小、成本低、尺寸小以及性能等方面的优点，成为了目前市场上最具成本效益的 ADC，也是最常见的 ADC。ADC 模块的精度一般有 8 位、10 位、12 位、16 位、24 位，例如 8 位精度就是 2 的 8 次方为 256 份，这里再加上一个参考值，相当于把这个参考值分成 256 等份。

8 位的精度：把 0~5V 分成 256 份，每份表示 5/256=0.02V;

10 位的精度：把 0~5V 分成 1024 份，每份表示 5/1024=0.005V;

12 位的精度：把 0~5V 分成 4096 份，每份表示 5/4096=0.0012V;

16 位的精度：把 0~5V 分成 65536 份，每份表示 5/65536=0.000076V;

24 位的精度：把 0~5V 分成 16777215 份，每份表示 5/16777215=0.00000023V;

这里的每份就相当于天平的砝码的最小砝码的重量。

举一个具体的例子，假如一个 8 位 A/D 转换器输入待测模拟量为 6.84V，基准参考电压设置为 10V，就相当于把 10V 分为 2 的 8 次方个等份，10V 除以 256 为 0.0390625V，这是每份的值，在这里 AD 经过测量 D7~D0 的输出值为：10101111，转换成 10 进制后是 175，即这个待测量值是把 10V 分成 256 等份后，它大约是 175 等份这么多，即 0.0390625*175=6.8359375V，与实际输入的模拟电压 6.84V 的相对误差仅为 0.06%。

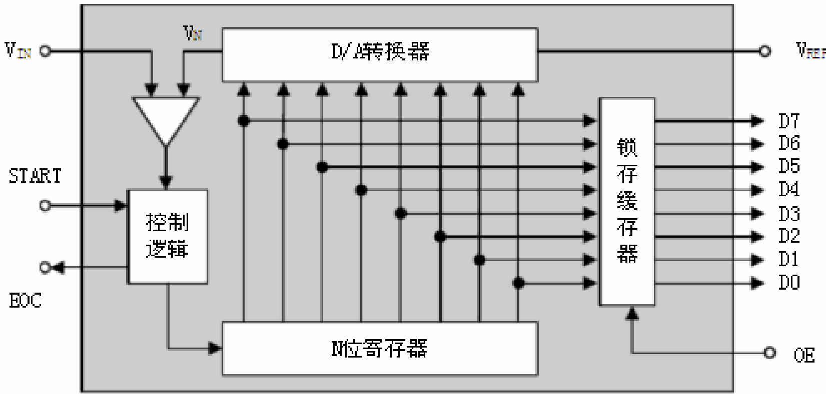


图 3-112 逐次逼近功能图

上图 3-112 是实际 AD 原理结构图，逐次逼近型 adc 由比较器、D/A 转换器、缓冲寄存器和若干控制逻辑电路构成。这里的 D0~D7 就是上面例子中的输出数据，原理是从高位到低位逐位比较，首先将缓冲寄存器各位清零；转换开始后，先将寄存器最高位置 1，把值送入 D/A 转换器，经 D/A 转换后的模拟量送入比较器，称为 Vo，与比较器的待转换的模拟量 Vi 比较，若 Vo<Vi，该位被保留，否则被清 0。然后，再置寄存器次高位为 1，将寄存器中新的数字量送 D/A 转换器，输出的 Vo 再与 Vi 比较，若 Vo<Vi，该位被保留，否则被清 0。循环此过程，直到寄存器最低位，得到数字量的输出。

在确定硬件地址（这个我们下面会有介绍到），与之通信后，芯片在做 AD 功能时，根据那模拟输入得到的模拟信号，芯片会经过 4 个步骤把模拟信号转换成数字信号，分别是：采样、保持、量化与编码。

1）采样

又称为抽样，将模拟信号抽样就是把一段模拟信号分解成许多节点；例如直线上扬的股市曲线图，昨天是 5000 点，今天是 6000 点，明天是 8000 点，这 3 个数字虽然无法描述每一天完整的曲线走势图，但可以大概描述出来大体的走势是一个上涨的趋势。如果把这个抽样再改进一下，变成昨天上午是 4500 点，昨天下午是 5000 点，今天上午是 5500 点，今天下午是 6000 点，明天上午是 7000 点，明天下午是 8000 点；那么可以看到这个曲线又更加的真实了。采样点增加了一倍，股市曲线走势图也清晰了不少。

2）保持

需要这个信号连续的保持一段时间，否则就有可能是一个干扰或者信号毛刺，这样的干扰和毛刺是不符合采样标准的。

用专业术语来描述就是后续的量化过程需要一定的时间 τ，对于随时间变化的模拟输入信号，要求瞬时采样值在时间 τ 内保持不变，这样才能保证转换的正确性和转换精度，这个过程就是采样保持。正是有了采样保持，实际上采样后的信号是阶梯形的连续函数。

3）量化

又称幅值量化，把采样信号经过舍入或截尾的方法变为只有有限个有效数字的数，这一过程称为量化。

若取信号可能出现的最大值 A，令其分为 D 个间隔，则每个间隔长度为 R=A / D，R 称为量化增量或量化步长。当采样信号落在某一小间隔内，经过舍入或截尾方法而变为有限值时，则产生量化误差。

一般又把量化误差看成是模拟信号作数字处理时的可加噪声，故而又称之为舍入噪声或截尾噪声。量化增量 D 愈大，则量化误差愈大，量化增量大小，一般取决于计算机 A/D 卡的位数。例如，8 位二进制为 28=256，即量化电平 R 为所测信号最大电压幅值的 1 / 256。

像上面举的股市曲线走势图中的 5000 点，6000 点，8000 点等都是已经被量化出来的值。

4）编码

将离散幅值经过量化以后变为二进制数字的过程，采样后的这些节点如何变成程序代码中可以识别的二进制码，如何把这些节点合理的顺序排列出来，这样就实现了把一连串的模拟信号用数字信号给描述出来的过程，这个数字信号就是编码。

这里要提一点，无论将模拟信号如何采样转换成数字信号，都有误差的，采样不可能那么全，所以数字信号只能描述模拟信号的绝大部分，不可能 100%的完全反馈出模拟信号，所以在电子产品中，很多模数转换芯片都是自己的精度的，精度就跟采样的精度有关系，这块大家接下来可以留意一下。

3.18.4 数模D/A的转换原理

D/A 转换器上有个模拟输出端口，它的作用和前面介绍的 A/D 相反，它是把数字信号转换成模拟信号；D/A 有很多类型的，这里介绍一种典型的 T 型电阻网络 D/A 转换器原理，如下图 3-113 所示：

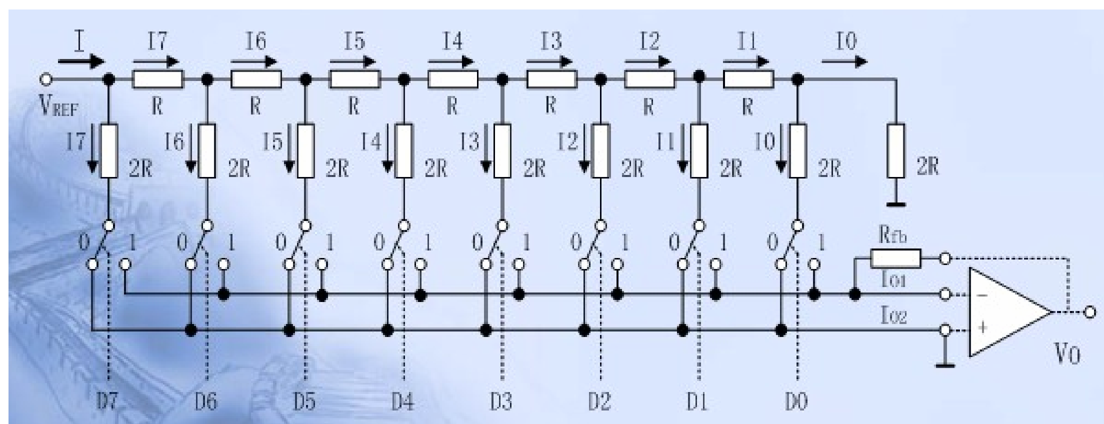


图 3-113 T 型电阻网络 D/A 转换器

由图可知，运放两个输入端为“虚地”，所以电位都约为 0。所以无论开关在 0 或者 1，最后两个 2R 都是并联得 R，和电阻 R 串联又为 2R,以此类推，那么到最前端，相当于两个 2R 的电阻并联，可知电流 I=Vref/R。I7=I/2,I6=1/2*I/2,由此追溯到 I0=I/256,如果 Rfb=R，那么 V0 只与 Vref 有关，即 V0=Vref*z/256。

数字量是用代码按数位组合起来表示的，对于有权码，每位代码都有一定的位权。为了将数字量转换成模拟量，必须将每 1 位的代码按其位权的大小转换成相应的模拟量，然后将这些模拟量相加，即可得到与数字量成正比的总模拟量，从而实现了数字—模拟转换。这就是组成 D/A 转换器的基本指导思想。

我们可以把 D/A 转换等效为下图，数字信号控制开关的闭合，如 1 为闭合开关，0 为打开开关，根据数据信号从而改变电路中 I 电流的大小，导致输出电压的变化，控制开关的数字信号由 I2C 提供，图中的 Uo 为芯片上的模拟输出。

3.18.5 A/D与D/A的主要指标

1．分辨率

ADC 的分辨率是指使输出数字量变化一个最小量时模拟信号的变化量。常用二进制的位数表示；例如 8 位的 AD，可以描述 255 个刻度的精度（2 的 8 次方），在它测量一个 5V 左右的电压时，它的分辨率是 5V 除以 255，大约为 0.02V 这样一个分辨率，也就是说它每改变一个 AD 的刻度，最小单位必须是 0.02V。

2．量化误差

ADC 把模拟量变为数字量，用数字量近似表示模拟量，这个过程称为量化。量化误差是 ADC 的有限位数对模拟量进行量化而引起的误差。

实际上，要准确表示模拟量，ADC 的位数需很大甚至无穷大。一个分辨率有限的 ADC 的阶梯状转换特性曲线与具有无线分辨率的 ADC 转换特性曲线（直线）之间的最大偏差即是量化误差。如下图 3-114：

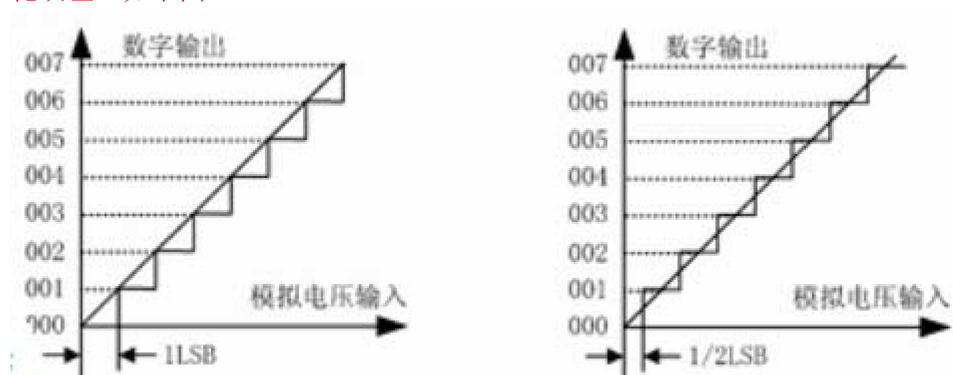


图 3-114 转换特性曲线

3. 偏移误差

偏移误差是指输入信号为零时，输出信号不为零的值，所以有时又称为零值误差。假定 ADC 没有非线性误差，则其转换特性曲线各阶梯中点的连线必定是直线，这条直线与横轴相交点所对应的输入电压值就是偏移误差。

4. 转换速率

转换速率是指完成一次从模拟转换到数字的所需要的时间的倒数。

积分型的 AD 的转换时间是毫秒级，属于低速 AD； 逐次比较型 AD 是微秒级的 AD，属于中速 AD；并行/串行的 AD 可达到纳秒级，属于高速的 A/D。

5. I2C 总线串行输入输出

因为 A/D 转换都是电路的比对，所以速度很快，主要瓶颈还是在 I2C 的速度上，I2C 无法达到那么高的速度把转换得来的值发送出去，因为 I2C 总线本身的速度是不高的。

6. 线性度

线性度有时又称为非线性度，它是指转换器实际的转换特性与理想直线的最大偏差

7. 绝对精度

在一个转换器中，任何数码所对应的实际模拟量输入与理论模拟输入之差的最大值，称为绝对精度。对于 ADC 而言，可以在每一个阶梯的水平中点进行测量，它包括了所有的误差。

3.18.6 PCF8591 芯片分析

PCF8591 是具有 I2C 总线接口 8 位精度的 A/D 及 D/A 转换器芯片，有 4 路 A/D 输入，1 路 D/A 模拟输出；也就是说，它既可以作 A/D 转换也可以作 D/A 转换；其中 A/D 转换为逐次比较型。

另外它还有 3 个地址引脚 A0、A1 和 A2 用于编程硬件地址，允许最多 8 个器件连接到 I2C 总线而不需要额外硬件；器件地址、控制和数据通道通过两线双向 I2C 总线传输。所以器件可以满足多路服用模拟量输入、片上跟踪和保持功能、8 位模数转换和 8 位数模转换，最大转换速率取决于 I2C 总线的最高速率。

PCF8591 引脚图如下图 3-115 所示：

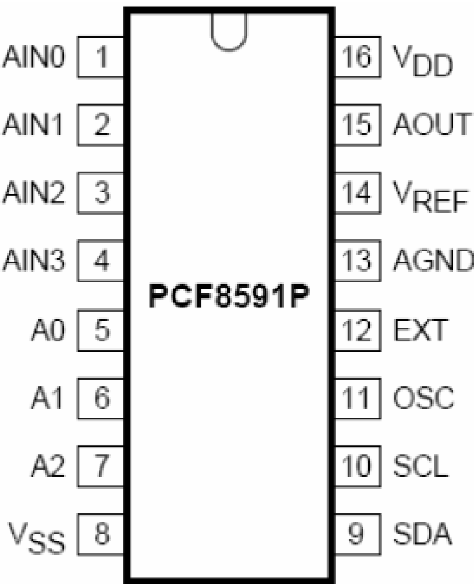


图 3-115 PCF8591 引脚图

PCF8591 引脚功能介绍如下表 3-72

表 3-72 PCF8591 引脚功能介绍

引脚	序号	引脚功能
AIN0	1	模拟量输入端
AIN1	2	
AIN2	3	
AIN3	4	
A0	5	引脚地址端
A1	6	
A2	7	
Vss	8	地
SDA/ SCL	9/10	I2C 总线的数据线和时钟线
OSC	11	外部时钟输入端，内部时钟输出端
EXT	12	内部、外部时钟选择线，使用内部时钟时 EXT 接地。
AGND	13	模拟信号地
Vref	14	参考电压输入
AOUT	15	模拟量输出
Vdd	16	电源

其中值得注意的是，VRef 就是使用内部参考电压，如果你要使用外部参考电压，就将外部参考电压接到 Vref 管脚，PCF8591 的操作电压范围 2.5V-6V。

这个参考电压跟 0-5V 的输入范围并不矛盾，内部的 2.5V 参考电压提供给你，你可以用也可以不用，如果你接外部参考电压那么它就把内部的关了。如果你想输入端能接受 0-5V，那你必需接一个外部的 5V 参考电压，如果用内部的参考电压，输入电压的有效范围就是 0-2.5V。

下图 3-116 是 PCF8951 的内部架构图，可以看到各自负责的功能模块。

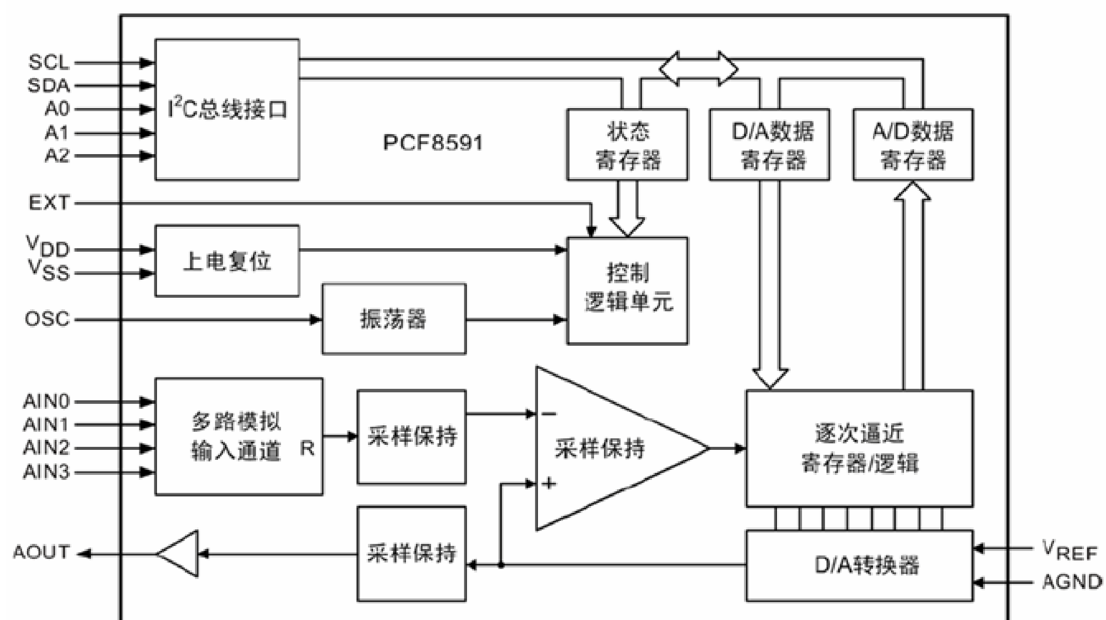


图 3-116 PCF8591 功能框图

PCF8591 芯片是 I2C 接口的，由于 51 单片机自身不带 I2C 接口，所以这里使用 P20 和 P21 两个 IO 管脚来模拟 I2C 总线，实现对 PCF8591 芯片的访问。PCF8591 与单片机的接口非

常简单，其中 A0，A1，A2 三个器件地址线都被拉到 GND，所以器件的地址始终都为 000。

3.18.6 PCF8591 芯片通信

第一个字节：

关于地址，因为 I2C 总线系统中的每一片 PCF8591 通过发送有效地址到该器件来激活，在 I2C 总线中允许最多挂载 8 个器件。I2C 中的设备地址包括固定部分和可编程部分。可编程部分必须根据地址引脚 A0、A1 和 A2 来设置，在 I2C 总线协议中地址必须是起始条件后作为第一个字节发送；地址字节的最后一位是用于设置以后数据传输方向的读/写位。



图 3-117PCF8591 地址位组成

第二个字节：

发送到 PCF8591 的第二个字节将被存储在控制寄存器，用于控制器件功能。控制寄存器的高板字节用于允许模拟输出，和将模拟输入编程为单端过查分输入。低半字节选择一个有高板字节定义的模拟输入通道。如果自动增量标志置 1，每次 A/D 转换后通道号将自动增加。

如果自动增量模式是使用内部振荡器的应用中所需要的，那么控制字中模拟输出允许标志应置 1。这要求内部振荡器持续运行，因此要防止振荡器启动延时的转换错误结果。模拟输出标志可以在其他时候复位以减少静态功耗。

PCF8591 芯片内部控制寄存器如下图：

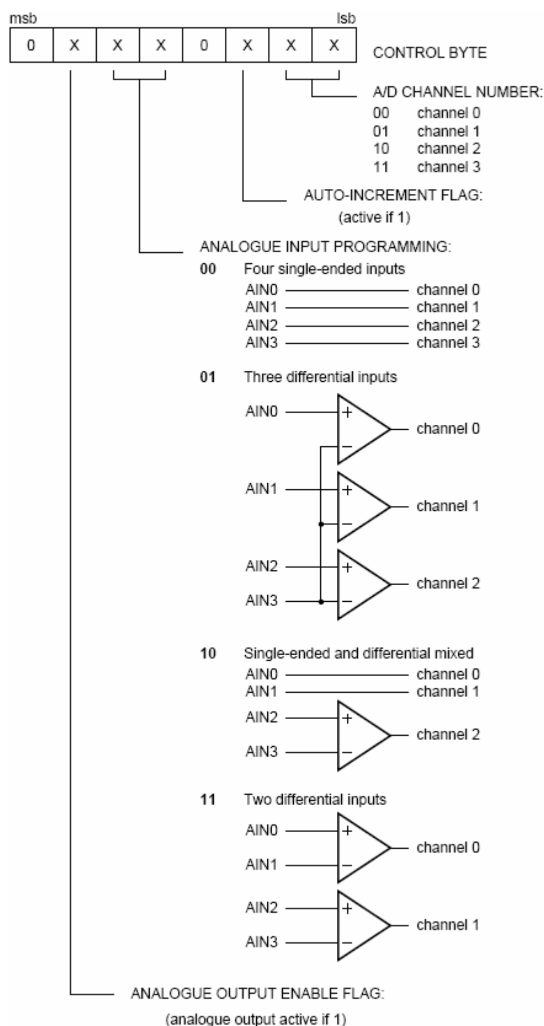


图 3-118　PCF8591 控制寄存器结构图

PCF8591 控制寄存器总共 8 位，第 1 位和第 2 位选择通道（总共有 4 个通道编号可选）；第 3 位是自动增益使能标志；第 4 位为空；第 5 位和第 6 位是选择模拟输入的模式：00 位四路单输入、01 位三路差分输入、10 位单端与差分配合输入、11 位模拟输出允许有效；第 7 位是当系统为 A/D 转换时，模拟输出使能标志为 1。

第三个字节：

在 D/A 转换里：发送给 PCF8591 的第三个字节被存储到 DAC 数据寄存器，并使用片上 D/A 转换器转换成对应的模拟电压。这个 D/A 转换器由连接至外部参考电压的具有 256 个（因为 D/A 是 8 位的，2 的 8 次方是 256）接头的电阻分压电路和选择开关组成。模拟输出电压由自动清零单位增益放大器缓冲。这个缓冲放大器可通过设置控制寄存器的模拟输出允许标志来开户或关闭。在激活状态，输出电压保持到新的数据字节被发送。

在 A/D 转换里：A/D 转换器采用逐次逼近转换技术。在 A/D 转换周期将来临时片上 D/A 抓换器和高增益比较器。一个 A/D 转换周期总是开始于发送一个有效读模式地址给 PCF8591 之后。A/D 转换周期在应答时钟脉冲的后沿被触发，并在传输前一次转换结果时执行。一旦一个转换周期被触发，所选通道的输入电压采样将保存到芯片被转换为对应的 8 为二进制码。

总结：

在进行数据操作时，首先是主控器发出起始信号，然后发出读寻址字节，被控器

做出应答后，主控器从被控器读出第一个数据字节，主控器发出应答，主控器从被控器读出第二个数据字节，主控器发出应答…一直到主控器从被控器中读出第 n 个数据字节，主控器发出非应答信号，最后主控器发出停止信号。

3.18.7 硬件原理图说明

神舟 51+ARM 开发板板载了与 PCF8591 芯片的硬件原理图如图 3-117 所示：

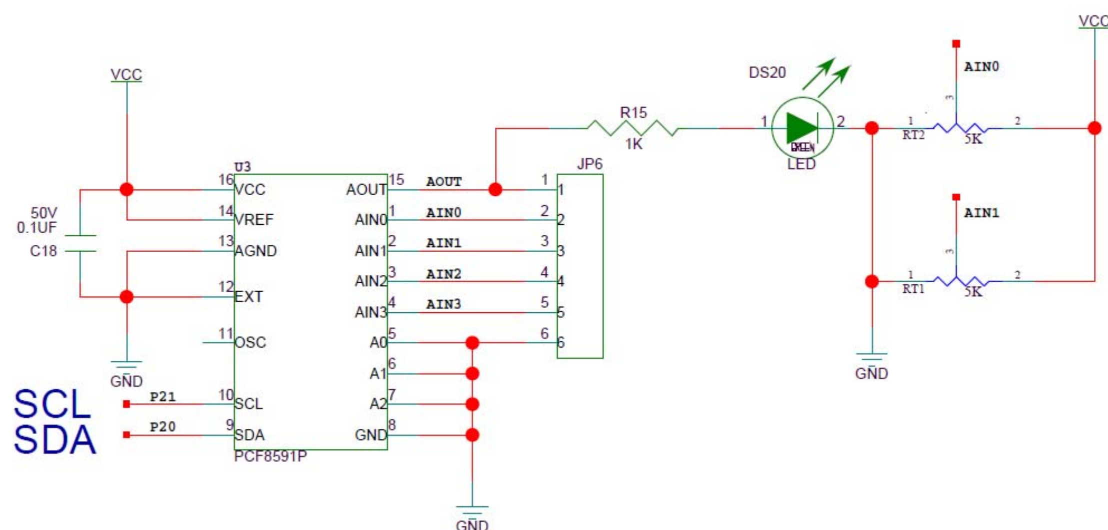


图 3-117PCF8591 硬件原理图

3.18.8 例程 01 PCF8591 第 1 路AD转换值数码管显示

本实验是通过 51 单片机的 IO 管脚模拟 I2C 协议访问 PCF8591 芯片。神舟 51 开发板上 PCF8591 第 1 路 AD 连接到了电位器 RT2，通过调节电位器 RT2 可以使 PCF8591 得到不同的模拟输入。将 PCF8591 转换值显示在数码管上动态显示。转换值范围为 0-255。实验调节电位器 RT2，看到数码管上的数值随之变化，则说明 PCF8591 访问成功。

程序如下：

```
/*************************************************************************
*  例程：PCF8591 第 1 路 AD 转换值数码管显示
*  作者：www.armjishu.com
*  版本：v1.0
*  内容：PCF8591 第 1 路 AD 转换值数码管显示，调节电位器 RT2 得到不同转换结果
*  现象：本实验是通过 51 单片机的 IO 管脚模拟 I2C 协议访问 PCF8591 芯片。神舟 51
*        开发板上 PCF8591 第 1 路 AD 连接到了电位器 RT2，转换值显示在数码管上
*        动态显示。转换值范围为 0-255。实验调节电位器 RT2，看到数码管上
*        的数值随之变化，则说明 PCF8591 访问成功。
*************************************************************************/
/* 包含头文件 */
```

```
#include <reg52.h>
#include "i2c.h"
#include "delay.h"
#include "display.h"
#define AddWr 0x90      //写数据地址
#define AddRd 0x91      //读数据地址
unsigned char ReadADC(unsigned char Chl);
/*------------------------------------------------
                  主程序
------------------------------------------------*/
main()
{
    unsigned char num=0;
    Init_Timer0();
    while (1)          //主循环
    {
        num=ReadADC(0);
        TempData[1]=DuanMa[num/100];
        TempData[2]=DuanMa[(num%100)/10];
        TempData[3]=DuanMa[(num%100)%10];
        //主循环中添加其他需要一直工作的程序
        DelayMs(100);
    }
}
/*------------------------------------------------
                  读 AD 转值程序
输入参数 Chl 表示需要转换的通道，范围从 0-3
返回值范围 0-255
------------------------------------------------*/
unsigned char ReadADC(unsigned char Chl)
{
    unsigned char Val;
    Start_I2c();               //启动总线
    SendByte(AddWr);         //发送器件地址 写命令
    if(ack==0)
    {
        return(0);
    }
    SendByte(0x40|Chl);       //发送控制字节
    if(ack==0)
    {
        return(0);
    }
    Start_I2c();
```

```
        SendByte(AddRd);        //发送器件地址 读命令
        if(ack==0)
        {
            return(0);
        }
        Val=RcvByte();              //读 AD 转值程序
        NoAck_I2c();               //发送非应位
        Stop_I2c();                //结束总线
        return(Val);
    }
```

由于PCF8591芯片在神舟51开发板中已经将SCL连接到51单片机的P21，SDA连接到51单片机的P20，所以只需要连接LED数码管。将神舟开发板的JP15排针连接到LED数码管的JP23。连接关系如表3-73

表3-73 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P0 口	JP15	直连	JP23	1 根 8 针扁平电缆	控制 LED 数码管
实验现象：调节电位器 RT2，看到数码管上的数值随之变化，则说明 PCF8591 访问成功。					

连接实物图如下图 3-118 所示：

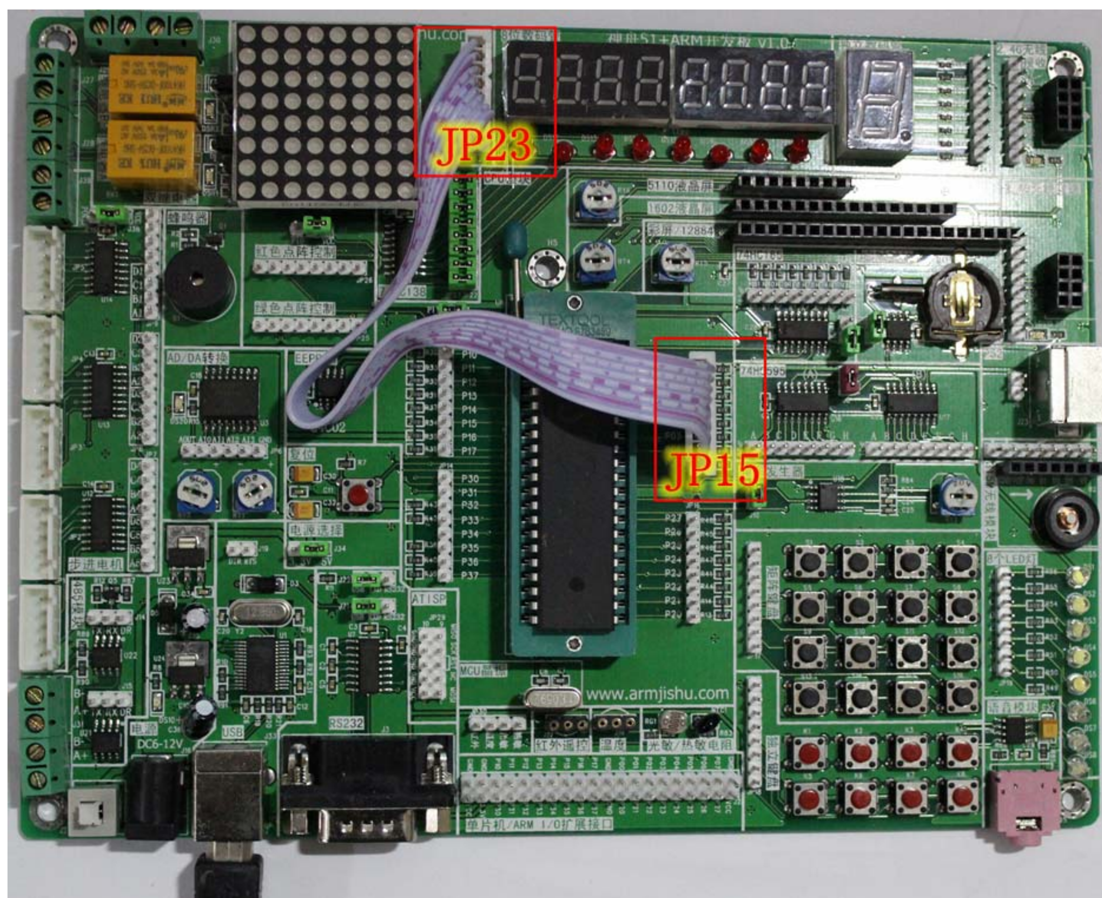


图 3-118 硬件连接实物图

知识要点：

1．因为 PCF8591 是 I2C 总线来访问的，所以它在 I2C 总现在中会有一个地址，该地址包括固定部分和可编程部分;可编程部分根据地址引脚 A0、A1 和 A2 来设置。其中 fixed part 是固定部分，programmable part 是可编程部分，最后一位 R/W 是读还是写的位；地址格式如下图 3-119：

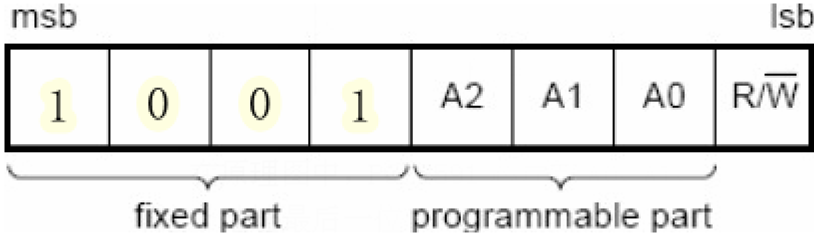


图 3-119 地址格式

在原理图中，PCF8591 芯片的 A0、A1、A2 三个地址都被拉低到 GND，表示 A0~A2 都为 0，那这个地址最后一位 R/W，如果是写 R/W=0，如果是读 R/W=1，这样就可以知道 PCF8591 芯片地址为 0x90 和 0x91 两个，一个是写，一个是读：

```
#define AddWr 0x90   //写数据地址
#define AddRd 0x91   //读数据地址
```

2.发送 SendByte（AddWr）是 I2C 发送写数据地址，SendByte（AddRr）是 I2C 发送读数据地址，具体函数实现可以看代码，按照 I2C 协议来走的，发给器件之后，器件稍后就会按照 I2C 协议来进行回应，反馈有效的数据。

3.下面解释一下 ReadADC()这个函数的执行流程：

3-1.首先通过函数 Start_I2c()启动 I2C 总线，然后调用 SendByte(AddWr)函数发送器件地址，告诉 PCF8591 这个器件将要它进行写操作。

3-2. SendByte(0x40|Chl)发送控制字节，这是发送的第 2 个字节，第一个字节是发送要写的地址,发送到 PCF8591 的第二个字节将被存储在控制寄存器,用于控制器件功能，控制寄存器的高半字节用于容许模拟输出，和将模拟输入编程为单端或差分输入。低半字节选择一个由高半字节定义的模拟输入通道。

那么我们分析一下写入的内容到底是什么意思？

----发送要写入的数据，可以看到如下图的寄存器说明，0x40|0 = 0x40,即下面的 D6 为 1，其它 D0~D5 以及 D7 都是 0;比如 D1 和 D0 都为 0，表示通道 0 输出，D5 和 D4 取 0 表示为四路单输入，这里的 AIN0 自动与通道 0 进行绑定了，就是说 AIN0=channel0。

----这个寄存器 PCF8591 器件的实际控制寄存器表如下表 3-74:

表 3-74 PCF8591 器件寄存器

寄存器位	D7	D6	D5	D4	D3	D2	D1	D0
内容	0	X	方式选择	方式选择	X	自动增益	A/D 通道	A/D 通道

- D1 和 D0 表示 A/D 通道数，D1 和 D0 的取值如下表 3-75：

表 3-75 D1、D0 真值表

D1	D0	通道
0	0	通道 0
0	1	通道 1
1	0	通道 2
1	1	通道 3

- D2 自动增益选择（有效位为 1）
- D5 和 D4 几种模拟量的输入方式选择，如表 3-76：

表 3-76 输入方式选择

D5	D4	**模拟量输入模式**	
0	0	四路单输入	00 Four single-ended inputs AIN0 — channel 0 AIN1 — channel 1 AIN2 — channel 2 AIN3 — channel 3
0	1	三路差分输入	01 Three differential inputs AIN0 + / AIN3 − → channel 0 AIN1 + / AIN3 − → channel 1 AIN2 + / AIN3 − → channel 2
1	0	单端与差分配合输入	10 Single-ended and differential mixed AIN0 — channel 0 AIN1 — channel 1 AIN2 + / AIN3 − → channel 2
1	1	两路差分输入	11 Two differential inputs AIN0 + / AIN1 − → channel 0 AIN2 + / AIN3 − → channel 1

这里就要提到一个新的概念，什么是差分输入？从严格意义上来讲，所有的信号都是差分信号，因为所有的电压只能是相对于另一个电压而言。大多数系统，我们都是把系统的 GND 作为基准点；对于 AD 讲的差分输入，就是把 2 个输入的差值作为最终信号的输入，例如下面这个图 3-120 就是差分信号，AIN0-AIN1 的值进行输出：

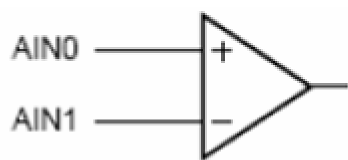


图 3-120 差分信号

差分信号的优点主要是抗干扰能力强，因为当外界存在噪声干扰时，几乎是同时被耦合到两条线上的，而接收端关心的只是两个信号的差值，所以外界的干扰几乎可以完全抵消，干扰使得电平变高就一起变高，要低就一起变低，所以这就是差分输入的优点所在。

如果不是差分信号，那么来一个干扰，这个信号就会变弱或者变强，影响到了最终接收端接收到的值。

3-3. 再次启动 I2C 总线，调用 Start_I2c()函数然后再发一个读命令读这个地址，SendByte(AddRd)其实就读第 1 路模拟输入通道 AIN0 的值;然后就通过代码 Val=RcvByte();来读取返回的 A/D 值，送到变量 Val。

3-4.最后通过主循环，把读回来的 A/D 值送给变量 num，然后再在数码管中显示出来

```
while (1)  //主循环
{
    num=ReadADC(0);
    TempData[1]=DuanMa[num/100];
    TempData[2]=DuanMa[(num%100)/10];
    TempData[3]=DuanMa[(num%100)%10];
    //主循环中添加其他需要一直工作的程序
    DelayMs(100);
}
```

3-5 关于 I2C 的时序，前面章节有讲解，这里暂时不做分析。

3.18.9 更多有关AD/DA的例程以及分析

更多 AD/DA 相应的例程可通过北京航空航天大学出版社下载专区下载，如下表 3-77：

表 3-77 A/D 和 D/A 更多丰富例程（含详细注释和文档分析）

序号	例程功能
例程 01	PCF8591 第 1 路 AD 转换值数码管显示
例程 02	PCF859 两路 AD 转换值数码管同时显示
例程 03	PCF8591 的 DA 输出控制 LED 亮度
例程 04	PCF8591 的 DA 输出三角波

3.19 RTC实时时钟（DS1302）

3.19.1 时钟

电子产品是如何记录时间的？实际是晶振发出稳定持续的振动波形，通过统计晶振的振动次数从而获得时间的，例如 1 秒钟会振动 32.768K 赫兹的晶振，只要能够测算到振动了这么多次，就算是过去了一秒钟时间，通过累计计算从而可以获得一分钟，一小时，甚至一天一月的时间。

常见的时钟芯片有 DS1302、DS1307、DS3231、PCF8485 等，它们可以记录时间，通过 51 单片机再把时钟芯片内部的数据读取出来，按照芯片规定的协议进行转化，就可以得到实际的当前时间。

3.19.2 DS1302 时钟芯片原理

DS1302 是一种高性能、低功耗(功耗小于 1mW)的实时时钟/日历芯片。它可以对年、月、日、周日、时、分、秒进行计时，每个月的天数和闰年的天数可自动调整，时钟操作可通过芯片内部寄存器 AM/PM 标志位决定采用 24 或 12 小时时间格式。

时钟芯片内部可以统计时间，自动计算出时间，这样就只需要 51 单片机取出时间信息就可以，所以时钟芯片上一定会有与 51 单片机进行通信的管脚；所有的工作离不开供电，对时钟芯片如果突然断电为了保证时钟不受影响，可能还需要加上切换到电池供电的功能。DS1302 的外部引脚功能说明如下图 3-121 所示：

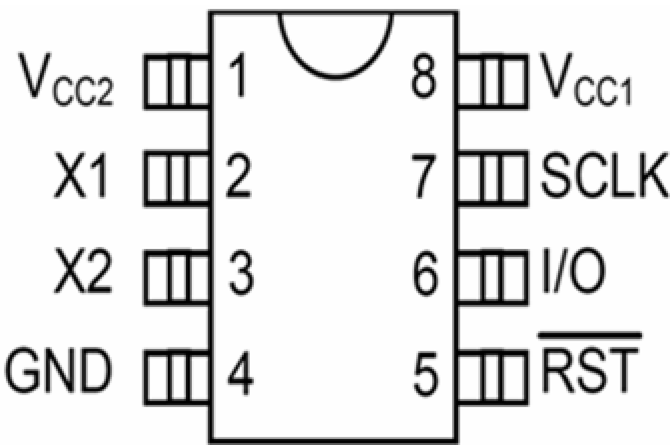


图 3-121 DS1302 外部引脚图

引脚功能说明如下表 3-78 所示：

表 3-78 DS1302 引脚功能说明

引脚	引脚说明
X1，X2	32.768K 晶振引脚
GND	地
RST	复位

I/O	数据输入/输出
SCLK	串行时钟
VCC1	电池引脚
VCC2	主电源引脚

关于电源。DS1302 含有主电源 VCC2/后背电源 VCC1 双电源引脚，同时提供了对后背电源进行涓细电流充电的能力。DS1302 的引脚排列,其中 VCC1 为后备电源，VCC2 为主电源。在主电源关闭的情况下，也能保持时钟的连续运行。DS1302 由 VCC1 或 VCC2 两者中的较大者供电。当 Vcc2 大于 VCC1＋0.2V 时，VCC2 给 DS1302 供电。当 VCC2 小于 VCC1 时，DS1302 由 VCC1 供电。X1 和 X2 是振荡源，外接 32.768kHz 晶振。RST 是复位/片选线，通过把 RST 输入驱动置高电平来启动所有的数据传送。

上电运行时，在 VCC<2.0V 之前，RST 必须保持低电平，直到 VCC 升到 2.0V 以上。并且只有在 SCLK 为低电平时，才能将 RST 置为高电平。

RST 信号。控制 DS1302 的是 RST 信号，拉低无效，拉高有效。RST 输入有两种功能：1）当 RST 为高电平时，允许对 DS1302 的 I/O 数据传送。2）RST 设置为低电平，终止 I/O 数据传送，I/O 引脚变为高阻态。

I/O 和 SCLK 信号。而 DS1302 的 I/O 是串行输入输出双向的，SCLK 为时钟输入端。写入一个字节和读取一个字节的时序不同，分为两种：1）串行输入。写入一个字节都是上跳沿有效。2）串行输出。一个字节先是上跳沿有效，然后下降沿有效。

3.19.3 DS1302 时钟芯片寄存器分析

DS1302 时钟芯片从内部和外部两个思路进行分析，内部主要描述芯片是如何记录和计算时间以及芯片内部资源是如何工作的；外部主要是时钟芯片如何与外部 51 单片机进行通信，使用什么指令来达到传送数据的目的。

对 DS1302 的操作就是对其内部寄存器的操作，DS1302 内部共有 12 个寄存器，其中有 7 个寄存器与日历、时钟相关，存放的数据位为 BCD 码形式。，如下表 3-79：

表 3-79 日历时钟相关寄存器

寄存器名	命令字节		范围	位内容							
	写	读		D7	D6	D5	D4	D3	D2	D1	D0
秒	80H	81H	00~59	CH	秒的十位			秒的个位			
分	82H	83H	00~59	0	分的十位			分的个位			
时	84H	85H	01~12 或 00~23	12/24	0	1/P	HR	小时个位			
日	86H	87H	01~31	0	0	日的十位		日的个位			
月	88H	89H	01~12	0	0	0	0/1	月的个位			
星期	8AH	8BH	01~07	0	0	0	0	0	星期几		
年	8CH	8DH	00~99	年的十位				年的个位			

表格解说：

1.秒钟寄存器；地址字节：0x80。秒钟寄存器除了记录秒钟以外，还控制了 DS1302 的时钟开关；秒寄存器的 CH 位，当写入 1 时 DS1302 停止工作，保持时间计时的最后一次状态；如果写入逻辑 0 时 DS1302 开始工作，时间从最后一次状态中继续计时。

为什么每一次写入秒钟，都会使 DS1302 工作呢？因为秒钟寄存器是八位寄存器，高四

位中的 D4~D6(D7 除外)记录最大可到 7，而低四位记录个位最大可到 15。而秒钟最大也就是 59，就算是换成十六进制最大也只需要记录 0x59。所以，最高位基本上是用不到的，但每次向秒钟寄存器进行初始化的时候都会很方便的把最高位 D7 也就是 CH 位初始化为 0，CH 位为逻辑 0，DS1302 就开始工作。

2.分钟寄存器；地址字节：0x82。八位寄存器，高四位记录十位(D7 除外)，低四位记录个位。

3. 时钟寄存器；地址字节：0x84。八位寄存器，高四位记录十位(D7 除外)，低四位记录个位。时钟寄存器的最高位，决定了时间是以 24 小时制，还是 12 小时。逻辑 0 是 24 小时制，逻辑 1 是 12 小时制。因为常规操作都是对寄存器初始化赋值为 0，所以 24 小时制是默认的；另外一个原因是时钟最大值是 23 小时，十六进制是 0x23，最高位也是用不到的，所以可以对其初始化赋 0 值。

4.日寄存器；地址字节：0x86。八位寄存器，高四位记录十位(实际上仅有 D4~D5 被使用)，低四位记录个位。

5.月寄存器；地址字节：0x88。八位寄存器，高四位记录十位(实际上仅有 D4 被使用)，低四位记录个位。

6.周寄存器；地址字节：0x8A。八位寄存器，仅有低四位被使用(D0~D3)，用来记录个位。

7.年寄存器；地址字节：0x8C。八位寄存器，高四位记录十位，低四位记录个位。

8.控制寄存器；地址字节：0x8E。八位寄存器，仅 D7 有用，D7 是 WP 位，逻辑 0 解除写保护，逻辑 1 开启写保护。换一句话说，每一次写其他寄存器 WP 位必须先置 0。

此外，控制寄存器、充电寄存器、时钟突发寄存器及与 RAM 相关的寄存器等，如下表 3-80：

表 3-80 其它寄存器及 RAM

寄存器名	命令字节		范围	位内容							
	写	读		D7	D6	D5	D4	D3	D2	D1	D0
写保护	83H	8FH	00H~80H	WP							
涓流充电	90H	91H	-	TCS				DS		RS	
时钟突发	8EH	BFH	-	-							
RAM 突发	FEH	FFH	-	-							
RAM0	C0H	C1H	00H~FFH	RAM 数据							
…	…	…	00H~FFH								
RAM30	FCH	FDH	00H~FFH								

备注：
1. WP：写保护位（置为 1 时，写保护；置为 0 时，未写保护）
2. TCS：1010 时慢充电；DS 为 01，选一个二极管，为 10，选 2 个二极管；11 或 00，禁止充电
3. RS：与二极管串联电阻选择。00，不充电；01，2KΩ 电阻；10，4KΩ 电阻；11，8KΩ 电阻

1. 涓流充电。DS1302 有后备供电的输入，亦即 VCC1 引脚，DS1302 允许透过控制内部的充电寄存器，经 VCC2 向 VCC1 流入的充电细流。当 VCC2 有电源输入时，VCC1 是停止供电的，但同一时间也可以为 VCC1 进行细流充电（这要看 VCC1 是否连接着可充电电池）。一旦 VCC2 停止供电，VCC1 就开始工作，与此同时细流充电就变成没有意义了。

2. RAM 突发。再一类为突发方式下的 RAM，此方式下可一次性读写所有的 RAM 的 31 个字节，命令控制字为 FEH（写）、FFH（读）。

3.DS1302 设立一个 31 个字节的空间，可以方便用户存放一些数据，前提是在不断电的情况下；DS1302 内部的 RAM 分为两类，一类是单个 RAM 单元，共 31 个，每个单元为一个 8

位的字节，其命令控制字为 COH~FDH，其中奇数为读操作，偶数为写操作。

3.19.4 DS1302 硬件连接原理

神舟 51 开发板板载了 DS1302 时钟芯片以及备用电池，无论神舟 51 开发板主电源是否有电都可以保证时间同步。

DS1302 与单片机的接口比较简单，DS1302 硬件连接原理图如下图 3-124 所示。

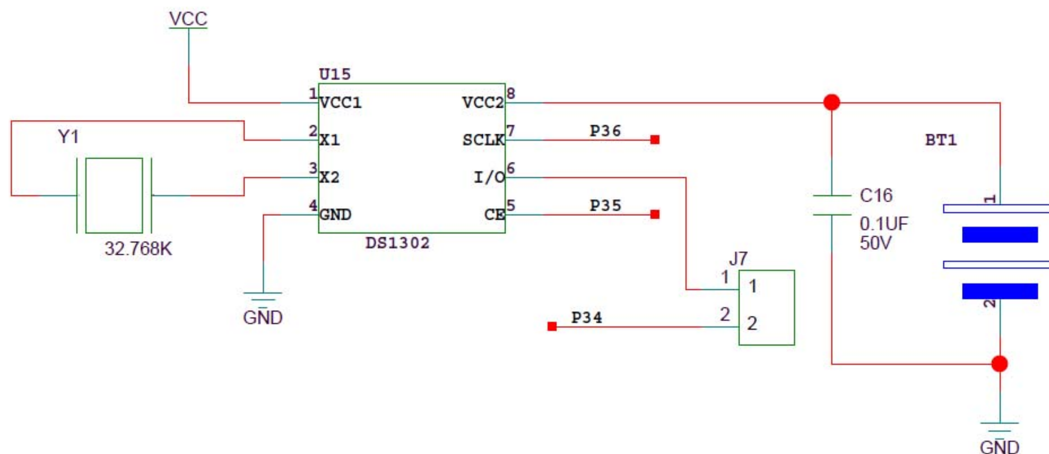


图 3-124 DS1302 硬件连接原理图

实验时请将将开发板上的 J7 跳冒接上，J10 上的跳冒跳开。

3.19.5 例程 01 DS1302 数码管显示实时时钟

代码如下：

```
/*********************************************************************
* 例程：DS1302时钟数码显示
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：将时间写入DS1302芯片，从DS1302读出时间在数码管上实时显示
* 现象：通过本例程了解DS1302时钟芯片的基本原理，理解并掌握DS1302时钟
*         芯片驱动程序的编写以及实现数字字符在数码管中的显示。
*         数码管的动态显示是由定时中断处理程序定时动态显示的。
*         LED数码管动态显示的时间每秒变化一次则说明DS1302芯片操作成功，
*         否则操作失败
* 注意：JP7跳线冒要短接。
*********************************************************************/
/* 包含头文件，一般情况不需要改动，头文件包含特殊功能寄存器的定义 */
#include <reg52.h>
#include <intrins.h>
sbit SCK = P3^6;        //时钟
```

```
sbit SDA = P3^4;          //数据
sbit RST = P3^5;          // DS1302复位
sbit LS138A = P2^2;    //定义138译码器的输入脚A由P2.2控制
sbit LS138B = P2^3;    //定义138译码器的输入脚B由P2.3控制
sbit LS138C = P2^4;    //定义138译码器的输入脚C由P2.4控制
bit ReadRTC_Flag;      //定义读DS1302标志

unsigned char l_tmpdate[7]={0,58,16,10,9,1,12};    //秒分时日月周年12-09-10 16:58:00
unsigned char l_tmpdisplay[8];
//秒分时日月周年  最低位为读写位  0为写  1为读
code unsigned char write_rtc_address[7]={0x80,0x82,0x84,0x86,0x88,0x8a,0x8c};
code unsigned char read_rtc_address[7] ={0x81,0x83,0x85,0x87,0x89,0x8b,0x8d};
//共阴数码管  0-9   '-' '_' '熄灭‘表
code unsigned char table[]={0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f,
                                                  0x40,0x08,0x00,0x00,0x00,0x00};
/******************************************************************/
/*                              函数声明                                  */
/******************************************************************/
void Write_Ds1302_byte(unsigned char temp);
void Write_Ds1302( unsigned char address,unsigned char dat );
unsigned char Read_Ds1302 ( unsigned char address );
void Read_RTC(void);//read RTC
void Set_RTC(void); //set RTC
void InitTIMER0(void);//inital timer0
/******************************************************************/
/*                             主函数                                  */
/******************************************************************/
void main(void)
{
    RST=0;
    SDA=0;
    InitTIMER0();       //初始化定时器0
    //写入时钟值，如果使用备用电池，则不需要每次上电写入，此程序应该屏蔽
    Set_RTC();
    while(1)
    {
        if(ReadRTC_Flag)
        {
            ReadRTC_Flag=0;
            Read_RTC();

            //数据的转换，我们采用数码管0~9的显示,将数据分开
            //时
            l_tmpdisplay[0]=l_tmpdate[2]/16;
```

```
                l_tmpdisplay[1]=l_tmpdate[2]&0x0f;
                //加入"-"间隔符
                l_tmpdisplay[2]=10;
                //分
                l_tmpdisplay[3]=l_tmpdate[1]/16;
                l_tmpdisplay[4]=l_tmpdate[1]&0x0f;
                //加入"-"间隔符
                l_tmpdisplay[5]=10;
                //秒
                l_tmpdisplay[6]=l_tmpdate[0]/16;
                l_tmpdisplay[7]=l_tmpdate[0]&0x0f;
            }
        }
    }
```

硬件连接关系如下表 3-82 所示：

表 3-82 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P0	JP15(A 向左)	直连	JP23(B 向左)	8 芯排线	控制 LED 数码管
P22-P24	JP27	跳帽	JP22	8 个跳帽	74LS138 数码管位选
由于 DS1302 芯片在神舟 51 开发板中已经将 CLK 连接到 51 单片机的 P3.6，RST/CE 连接到 51 单片机的 P3.5，I/O 通过跳线 J7 连接到 51 单片机的 P3.4。所以只需将调帽短接 J7，然后要连接 LED 数码管。将神舟开发板的 JP15 排针连接到 LED 指示灯的 JP23。					
实验现象：如果 LED 数码管动态显示的时间每秒变化一次则说明 DS1302 芯片操作成功，否则操作失败。					

硬件实物图连接如下图 3-125：

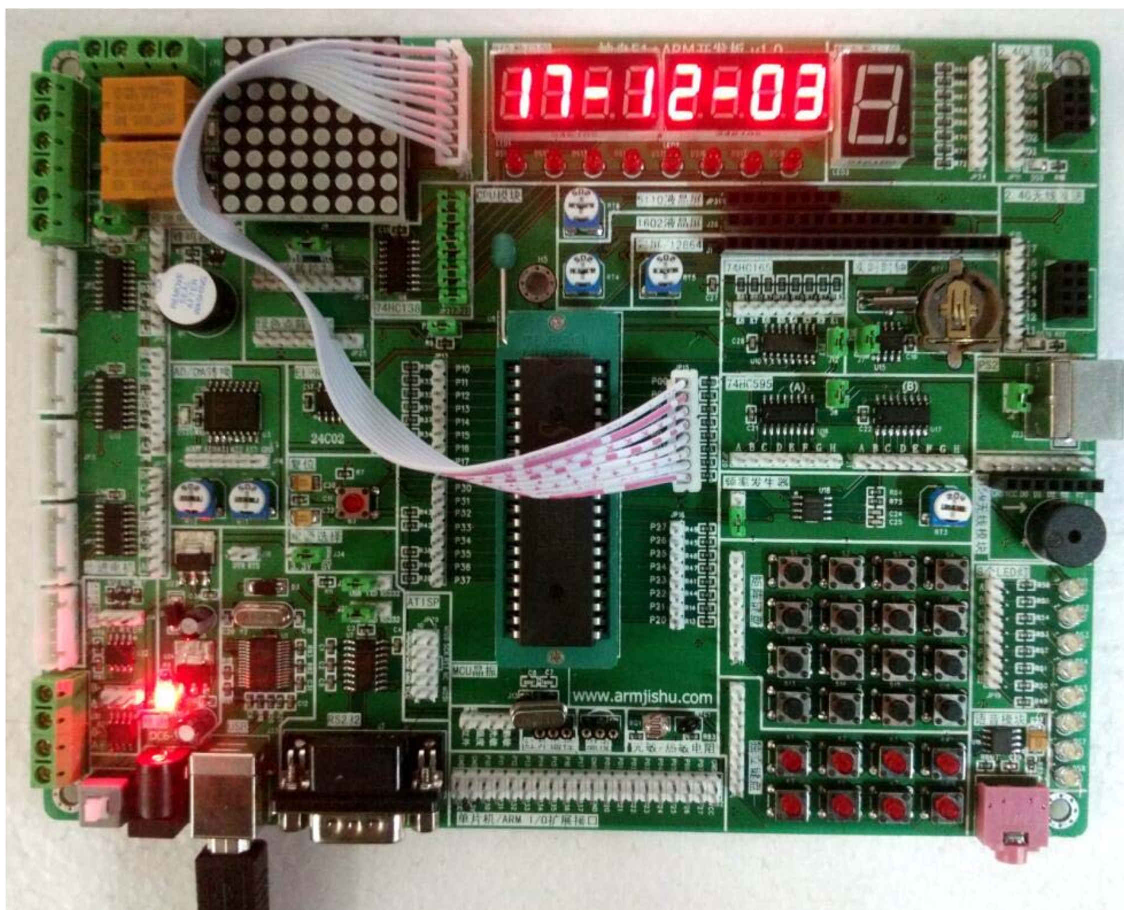


图 3-125 硬件连接实物图

知识要点：

1.本实验是通过 51 单片机的 IO 管脚模拟 DS1302 的协议访问 DS1302 芯片设置和读取实时时钟。实验首先向 DS1302 芯片写入默认时间“2012-09-10 16:58:00”，然后定时将时间读出显示在 LED 数码管上。LED 数码管上显示时、分、秒 的实时值。LED 数码管动态显示的时间每秒变化一次则说明 DS1302 芯片操作成功，否则操作失败。

2. Write_Ds1302_Byte()函数和 Write_Ds1302()函数的分析，代码如下：

```
void Write_Ds1302_Byte(unsigned  char temp)
{
    unsigned char i;
    for (i=0;i<8;i++)     //循环 8 次写入数据
    {
        SCK=0;
        SDA=temp&0x01;   //每次传输字节的最低比特
        SCK=1;
        temp>>=1;        //右移一位
    }
}
```

SCLK 是时钟线（程序中是 SCK 变量），I/O 是数据线（程序中是 SDA 变量）。请注意数据是对时钟信号敏感的，而且一般数据是在上升沿写入，所以程序中的 SCK 首先为 0，传输数据的时候变成 1，使得 SDA 在上升沿的时候传输数据，可以看到写操作时序如下图 3-112：

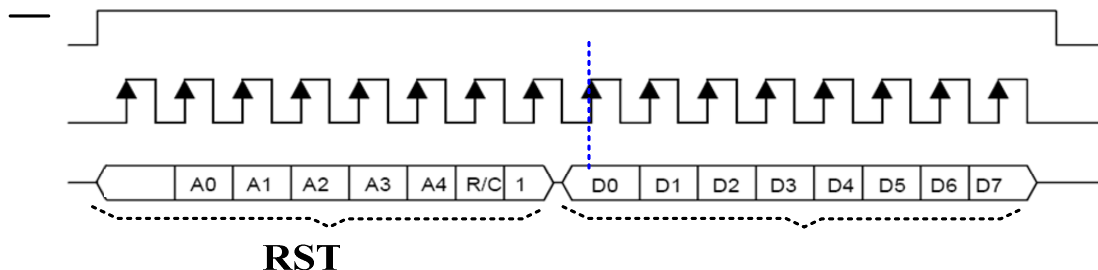


图 3-122 写操作时序

再分析 Write_Ds1302（）函数，可以看到，其中 RST 从低电平变成高电平启动一次数据传输过程，传输完数据 RST 又变成低电平(RST=0)。

```
void Write_Ds1302( unsigned char address,unsigned char dat )
{
    RST=0;
    SCK=0;
    SDA=0;
    RST=1;                          // 启动
    Write_Ds1302_Byte(address); // 发送地址
    Write_Ds1302_Byte(dat);      // 发送数据
    RST=0;                          // 恢复
}
```

3. char Read_Ds1302_Byte()读函数分析，代码如下：

```
unsigned char Read_Ds1302_Byte(void)
{
    unsigned char i, readdata = 0;
    for (i=0;i<8;i++)              // 循环 8 次，读取数据
    {
        SCK=0;
        readdata>>=1;              // 右移一位
        if(SDA)
        {
            readdata|=0x80;      // 采样数据
        }
        SCK=1;
    }
    return readdata;
}
```

读函数的数据 readdata 右移，第一位读到的是最低位，逐渐从 D0 读到 D7 总共 8 次循环，如果判断 SDA 数据线为高电平即 SDA=1，即对 readdata 对应 bit 位置 1，否则直接右移，对应位补 0，读操作时序如下图 3-123 所示：

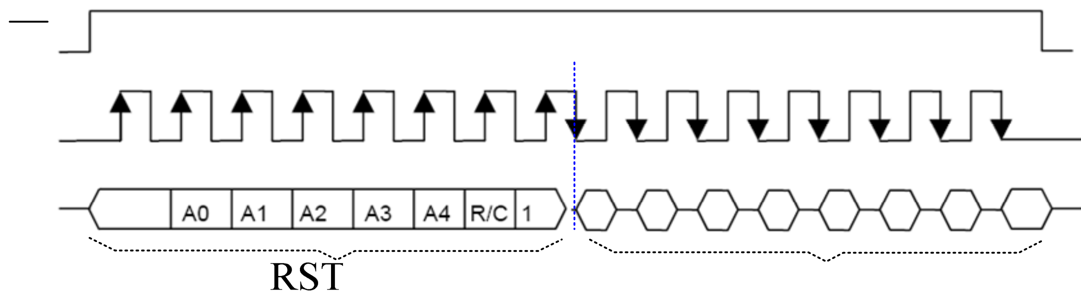


图 3-123 读操作时序

Read_Ds1302（）函数是读 DS1302 芯片的指定寄存器位置的函数，按照上图的时序把地址写入进去，然后等待下一个 SCLK 时钟返回对应数据，用 readdata 变量返回，读取过程 RST 要为高电平，读取完毕 RST 变为低电平。

```
unsigned char Read_Ds1302 ( unsigned char address )
{
    unsigned char readdata =0x00;
    RST=0;
    SCK=0;
    SDA=0;
    RST=1;
    Write_Ds1302_Byte(address);
    readdata = Read_Ds1302_Byte();
    SCK=1;
    RST=0;
    RST=0;
    return readdata;  // 返回 8 位数据
}
```

4. 单片机是通过简单的同步串行通讯与 DS1302 通讯的，每次通讯都必须由单片机发起，无论是读还是写操作，单片机都必须先向 DS1302 写入一个命令帧，这个帧的格式如下表 3-77 所示，最高位 BIT7 固定为 1，BIT6 决定操作是针对 RAM 还是时钟寄存器，接着的 5 个 BIT 是 RAM 或时钟寄存器在 DS1302 的内部地址，最后一个 BIT 表示这次操作是读操作抑或是写操作。

表 3-77 命令帧格式

BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
1	R/C	A4	A3	A2	A1	A0	R/W
固定为 1	为 0 时选择操作时钟，为 1 时选择操作 RAM	操作地址					为 0 时进行写操作，为 1 时进行读操作

DS1302 工作时为了对任何数据传送进行初始化，需要将复位脚（RST）置为高电平且将 8 位地址和命令信息装入移位寄存器。数据在时钟（SCLK）的上升沿串行输入，前 8 位指定访问地址，命令字装入移位寄存器后，在之后的时钟周期，读操作时输出数据，写操作时输出数据。时钟脉冲的个数在单字节方式下为 8+8（8 位地址+8 位数据），在多字节方式下最多可达 248 的数据（8 位地址+248 位数据）

5. Read_RTC（）函数读取 DS1302 芯片中的日历数据

```
void Read_RTC (void)
```

```
{
    unsigned char i,*p;
    p=read_rtc_address;          //地址传递
    for(i=0;i<7;i++)             // 分 7 次读取，秒分时日周月年
    {
        l_tmpdate[i]=Read_Ds1302(*p);
        p++;
    }
}
```

read_rtc_address 是全局变量的数组，里面存放着对应的秒分时日周月年的寄存器地址，数组代码如下：

code unsigned char read_rtc_address[7] ={0x81,0x83,0x85,0x87,0x89,0x8b,0x8d};

可以在上面知道 0x81 地址就是秒的读寄存器，0x83 就是分的读寄存器地址，以此类推可以一直读到年；将该地址通过函数 Read_Ds1302()来读取对应的值，存入数据 l_tmpdate[i]中，就把秒分时日周月年这个七个数据给读出来并保存起来了。

6．Set_RTC（）设置初始时间，先把char l_tmpdat[]数组中的值转化成BCD码，方便将该值设置到DS1302芯片中去，具体数组如下：

unsigned char l_tmpdate[7]={0,58,16,10,9,1,12};//设置初试时间为12-09-10 16:58:00

紧接着，分别把这些时间设置到对应的秒分时日周月年的寄存器中去，具体的命令如下：

code unsigned char write_rtc_address[7]={0x80,0x82,0x84,0x86,0x88,0x8a,0x8c};

这个命令可以查看之前的DS1302寄存器表格，这些写命令；接着对时钟突发寄存器0x8E写入0x00，就是允许写操作Write_Ds1302(0x8E,0X00);如果是Write_Ds1302(0x8E,0X80);就表示禁止写操作。因为要设置日历，所以必须设置DS1302进入可写状态。

```
void Set_RTC (void)          //设定 日历
{
    unsigned char i,*p,tmp;
    //设置DS1302时间
    for(i=0;i<7;i++)
    {   //将默认时间转换为BCD码
        tmp=l_tmpdate[i]/10;
        l_tmpdate[i]=l_tmpdate[i]%10;
        l_tmpdate[i]=l_tmpdate[i]+tmp*16;
    }
    Write_Ds1302(0x8E,0X00); //允许写操作
    p=write_rtc_address;     //传地址
    for(i=0;i<7;i++)         //7次写入 秒分时日月周年
    {
        Write_Ds1302(*p,l_tmpdate[i]);
        p++;
    }
    Write_Ds1302(0x8E,0x80); //禁止写操作
}
```

7. 对定时器 0 进行初始化，每隔一段时间就进入中断函数，中断函数负责现实从 DS1302 中读出时间参数显示到数码管中；因为人眼的特点，数码管虽然是每隔 2s 显示一次，但并

不影响显示效果，给人眼的感觉数码管一直在点亮状态中。

```
/******************************************************************/
/*                    定时器0初始化                                 */
/******************************************************************/
void InitTIMER0(void)
{
    TMOD|=0x01;                //定时器设置 16位
    TH0=(65536-2000)/256;      //定时 2ms
    TL0=(65536-2000)%256;
    ET0=1;
    TR0=1;
    EA=1;
}
/******************************************************************/
/*                    定时器中断函数                                */
/* 动态数码管显示，并隔段时间将标志位设为有效以便main读取1302的数据 */
/******************************************************************/
void tim(void) interrupt 1 using 1          //中断，用于数码管扫描
{
    static unsigned char i,num;
    TH0=(65536-2000)/256;                  //重新赋值 2ms
    TL0=(65536-2000)%256;
    //消隐，清空数据，防止有交替重影，改变数码位的瞬间全灭
    P0=0;
    switch(i)
    {
        case 0:LS138A=0; LS138B=0; LS138C=0; break;   //第1个数码管
        case 1:LS138A=1; LS138B=0; LS138C=0; break;   //第2个数码管
        case 2:LS138A=0; LS138B=1; LS138C=0; break;   //第3个数码管
        case 3:LS138A=1; LS138B=1; LS138C=0; break;   //第4个数码管
        case 4:LS138A=0; LS138B=0; LS138C=1; break;   //第5个数码管
        case 5:LS138A=1; LS138B=0; LS138C=1; break;   //第6个数码管
        case 6:LS138A=0; LS138B=1; LS138C=1; break;   //第7个数码管
        case 7:LS138A=1; LS138B=1; LS138C=1; break;   //第8个数码管
    }
    P0=table[l_tmpdisplay[i]];           //查表法得到要显示数字的数码段
    i++;
    if(i==8)
    {
        i=0;
        num++;
        //隔段时间读取1302的数据。时间间隔可以调整
        if(20==num)
        {
```

```
                //标志位置为有效
                ReadRTC_Flag=1;
                num=0;
            }
        }
    }
```

3.19.9 更多DS1302 实时时钟的例程以及分析

更多 DS1302 实时时钟相应的例程可通过北京航空航天大学出版社下载专区下载，如下表 3-83：

表 3-83 RTC 时钟（DS1302）更多丰富例程（含详细注释和文档分析）

序号	例程功能
例程 01	DS1302 数码管显示实时时钟
例程 02	DS1302 数码管显示实时时钟并断电保持
例程 03	DS1302 数码管显示实时时钟年月日
例程 04	DS1302 数码管和串口显示实时时钟年月日
例程 05	DS1302 时钟串口显示与设置
例程 06	1602 液晶与串口显示与设置 DS1302 时钟

3.20 1602 液晶屏

3.20.1 1602 字符型液晶屏简介

1）1602 液晶屏功能、显示方式等

液晶显示屏的英文名是Liquid Crystal Display，简称LCD。在日常生活中，我们对液晶显示器并不陌生。液晶显示模块已作为很多电子产品的通过器件，如在计算器、万用表、电子表及很多家用电子产品中都可以看到。液晶显示屏具有体积小、重量轻、功耗低等优点，所以LCD日渐成为各种便携式电子产品的理想显示器。下图为1602字符型液晶实物图：



图3-126 1602字符型液晶屏

根据 LCD 的显示内容划分，可以分为段式 LCD、字符式 LCD 和点阵式 LCD 3 种。其中，字符式 LCD 以其廉价、显示内容丰富、美观、使用方便等特点，成为 LED 数码管的理想替代品。1602 液晶屏就是字符式 LCD 的一种。字符型液晶显示是一种专门用于显示字母、数字、符号等字符式 LCD

2）1602LCD 的分类

1602LCD 分为带背光和不带背光两种，基控制器大部分为 HD44780，带背光的比不带背光的厚，是否带背光在应用中并无差别，两者尺寸差别如下图 3-127 所示：

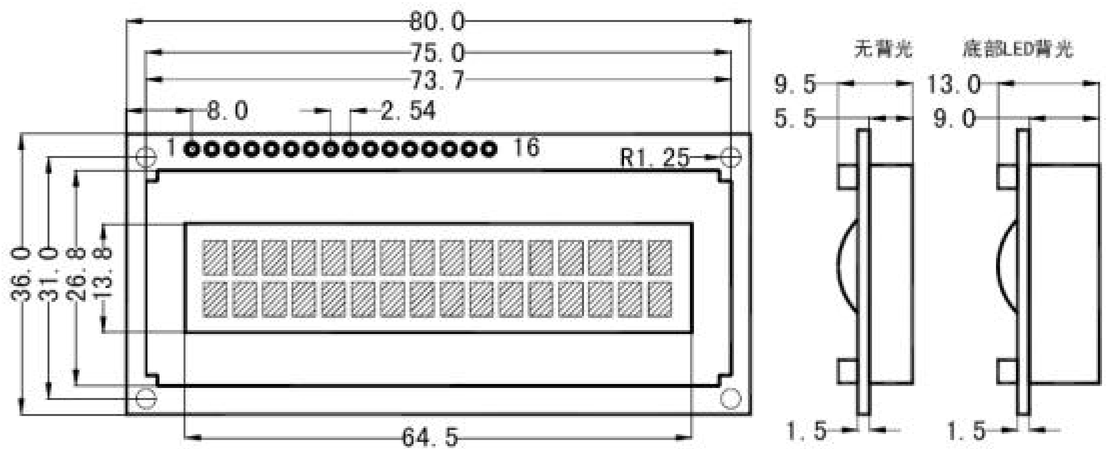


图 3-127 带背光与不带背光尺寸差别对比

3）功能管脚的介绍，如下表 3-84：

表 3-84 1602 控制器功能管脚介绍

编号	符号	引脚说明	编号	符号	引脚说明
1	VSS	电源地	9	D2	数据
2	VDD	电源正极	10	D3	数据
3	VL	液晶显示偏压	11	D4	数据
4	RS	数据/命令选择	12	D5	数据
5	R/W	读/写选择	13	D6	数据
6	E	使能信号	14	D7	数据
7	D0	数据	15	BLA	背光源正极
8	D1	数据	16	BLK	背光源负极

第 1 脚：VSS 为地电源。

第 2 脚：VDD 接 5V 正电源。

第 3 脚：VL 为液晶显示器对比度调整端，接正电源时对比度最弱，接地时对比度最高，对比度过高时会产生“鬼影”，使用时可以通过一个 10K 的电位器调整对比度。

第 4 脚：RS 为寄存器选择，高电平时选择数据寄存器、低电平时选择指令寄存器。

第 5 脚：R/W 为读写信号线，高电平时进行读操作，低电平时进行写操作。当 RS 和 R/W 共同为低电平时可以写入指令或者显示地址，当 RS 为低电平 R/W 为高电平时可以读忙信

号，当 RS 为高电平 R/W 为低电平时可以写入数据。
第 6 脚：E 端为使能端，当 E 端由高电平跳变成低电平时，液晶模块执行命令。
第 7～14 脚：D0～D7 为 8 位双向数据线。
第 15 脚：背光源正极。
第 16 脚：背光源负极。

4）1602 液晶屏的特性与参数

1602 液晶屏的特性：

- +5V 电压，对比度可调
- 内含复位电路
- 提供各种控制命令，如：清屏、字符闪烁、光标闪烁、移位等多种功能
- 有 80 字节显示数据存储器 DDRAM
- 内建有 160 个 5x7 的字符发生器 CGROM
- 8 个可由用户自定义的 5x7 的字符发生器 CGRAM

1602 型 LCD 的主要技术参数如下：

- 显示容量：16X2 个字符
- 芯片工作电压：4.5V~5.5V
- 工作电流：2.0mA(5.0V)
- 模块最佳工作电压：5.0V
- 字符尺寸：2.95X4.35（WXH）mm

5）液晶常用的三种连接方式

常用液晶模块连接方式及其适用范围：
（1）金属插脚
金属引脚可直接焊接在 PCB（印刷线路板）上， 连接可靠，抗震动强，脚间距受限制 适用于音响产品，电表等。
（2）斑马纸(热封)
柔软性连接，组装不方便。 用于薄型产品的连接。 适用于计数器、寻呼机、电子记事簿等。
（3）导电橡胶
成本较低，组装方便， 是较多采用的连接方式 成本较低。 适用于手表，游戏机，时钟，电话等。
我们的 1602LCD 属于第一种连接方式，直接通过金属引脚连接

3.20.2 1602LCD显示的基本原理：

液晶显示的原理是利用液晶的物理特性，通过电压对显示区域进行控制，只要输入所需的控制电压，就可以显示出字符。线段的显示：点阵图形式液晶由 M×N 个显示单元组成，假设 LCD 显示屏有 64 行，每行有 128 列，每 8 列对应 1 字节的 8 位，即每行由 16 字节，共 16×8=128 个点组成，屏上 64×16 个显示单元与显示 RAM 区 1024 字节相对应，每一字节的内容和显示屏上相应位置的亮暗对应。例如屏的第一行的亮暗由 RAM 区的 000H——00FH 的 16 字节的内容决定，当（000H）=FFH 时，则屏幕的左上角显示一条短亮线，长度

为 8 个点；当（3FFH）=FFH 时，则屏幕的右下角显示一条短亮线；当（000H）=FFH，（001H）=00H，（002H）=00H，……（00EH）=00H，（00FH）=00H 时，则在屏幕的顶部显示一条由 8 段亮线和 8 条暗线组成的虚线。要 LCD 显示字符“A”，则只需将 A 的 ASCII 码 41H 存入 DDRAM，控制线路就会通过 HD44780 的另一个部件字符产生器（CGROM）将 A 的字型点阵数据找出来显示在 LCD 上。 这就是 LCD 显示的基本原理。。

字符的显示：

用 LCD 显示一个字符时比较复杂，因为一个字符由 6×8 或 8×8 点阵组成，既要找到和显示屏幕上某几个位置对应的显示 RAM 区的 8 字节，还要使每字节的不同位为“1”，其它的为“0”，为“1”的点亮，为“0”的不亮。这样一来就组成某个字符。但由于内带字符发生器的控制器来说，显示字符就比较简单了，可以让控制器工作在文本方式，根据在 LCD 上开始显示的行列号及每行的列数找出显示 RAM 对应的地址，设立光标，在此送上该字符对应的代码即可。

汉字的显示（点阵 LCD 实现）：

汉字的显示一般采用图形的方式，事先从微机中提取要显示的汉字的点阵码（一般用字模提取软件），每个汉字占 32B，分左右两半，各占 16B，左边为 1、3、5……右边为 2、4、6……根据在 LCD 上开始显示的行列号及每行的列数可找出显示 RAM 对应的地址，设立光标，送上要显示的汉字的第一字节，光标位置加 1，送第二个字节，换行按列对齐，送第三个字节……直到 32B 显示完就可以 LCD 上得到一个完整汉字

3.20.3 如何控制 1602 液晶屏（寄存器的介绍）

如何才能让液晶屏显示字符呢？用三个过程即可完成：

- 告诉液晶屏你应该双行或者单行显示，在哪里显示，显示的是 5*7 的字符还是 5*10 的字符等等
- 让液晶屏干什么的也就是液晶屏的命令。我们称这一过程为写命令。
- 告诉液晶屏显示什么字符，这一过程称为写数据。

要完成上面的操作，我们首先要先了解 LCD1602 的各个寄存器，并灵活的使用它们。

1）１６０２ＬＣＤ各寄存器介绍

控制器主要由指令寄存器ＩＲ、数据寄存器ＤＲ、忙识别位ＢＦ、地址计数器ＡＣ、ＤＤＲＡＭ、ＣＧＲＯＭ、ＣＧＲＡＭ及时序发生电路组成。下面我们详细的来了解它们。

- **指令寄存器（ＩＲ）和数据寄存器（ＤＲ）**

1602 液晶屏内部具有两个 8 位寄存器：指令寄存器（ＩＲ）和数据寄存器（DR），用户可以通过ＲＳ和Ｒ／Ｗ输入信号组合选择指定的寄存器，进行相应操作，它们组合选择方式如下表 3-85：

表 3-85 RS 和 R/W 输入信号组合

E	RS	R/W	说明
1	0	0	将 DB0——DB7 的指令代码写入到指令寄存器中
1 → 0	0	1	分别将状态标示 BF 和地址计数器(AC)内容读到 DB7 和 DB6
1	1	0	将 DB0——DB7 的数据写入到数据寄存器中，模块内容操作自动将数据写到 DDRAM 或 CGRAM 中
1 → 0	1	1	将数据寄存器内的数据读到 DB0——DB7，模块的内部操作

			自动将 DDRAM 或 CGARM 中的数据送入数据寄存器中

ＲＳ、Ｒ／Ｗ工作时序图如下图 3-128、3-129 所示：

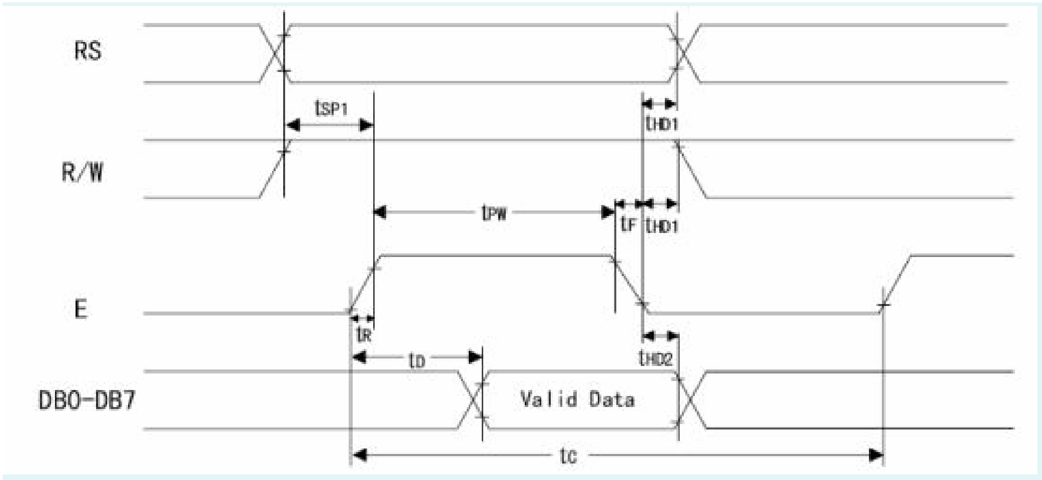


图 3-128 写操作时序

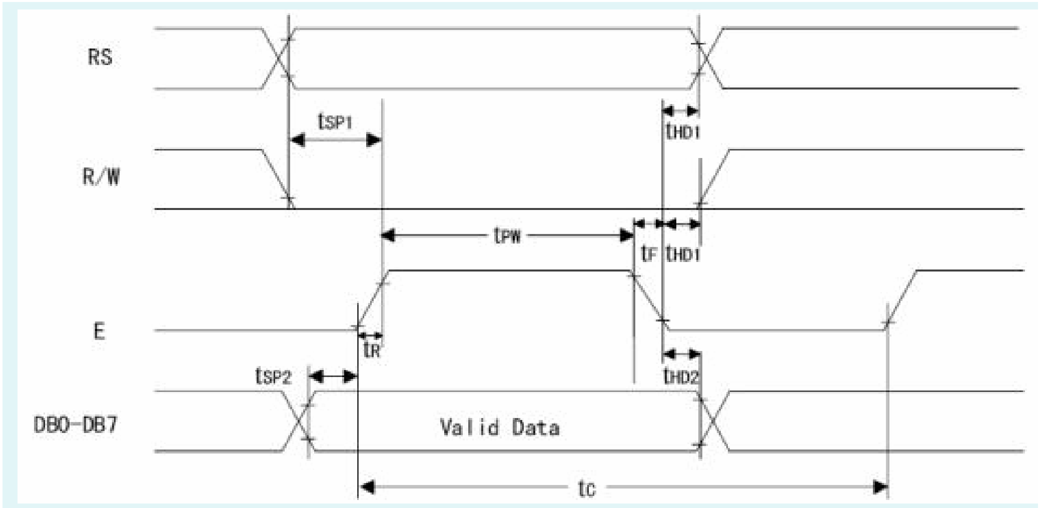


图 3-129 读操作时序

- **忙标识位ＢＦ**

忙标识为ＢＦ＝１时，表明模块正在进行内部操作，此时不接收任何外部指令和数据，当ＲＳ＝０,、Ｒ／Ｗ＝１且为高电平时，ＢＦ输出到ＤＢ７。每次最好先进行状态字检测，只有在确认ＢＦ＝０之后，ＭＰＵ才能访问模块。

- **地址计数器（ＡＣ）**

ＡＣ地址计数器是ＤＤＲＡＭ或ＣＧＲＡＭ的地址指针（ＤＤＲＡＭ或ＣＧＲＡＭ寄存器我们下面会有介绍）。随着ＩＲ中指令码的写入，指令码中携带的地址信息自动送入ＡＣ中，并做出ＡＣ作为ＤＤＲＡＭ的地址指针还是ＣＧＲＡＭ的地址指针的选择。

ＡＣ具有自动加１或自动减１的功能。当ＤＲ与ＤＤＲＡＭ或ＣＧＲＡＭ之间完成一次数据传送后，ＡＣ自动会加１或减１。在ＲＳ＝０、Ｒ／Ｗ＝１且Ｅ为高电平时，ＡＣ的内容送到ＤＢ６——ＤＢ０。地址计数器ＡＣ如下表 3-86 所示。

表 3-86 地址计数器ＡＣ

AC 高 3 位	AC 低 4 位

AC6	AC5	AC4	AC3	AC2	AC1	AC0

- **显示数据寄存器（ＤＤＲＡＭ）**

ＤＤＲＡＭ存储显示字符的字符码，其容量的大小决定模块最多可显示的字符数目。控制器内部有８０字节的ＤＤＲＡＭ缓冲区，ＤＤＲＡＭ地址与ＬＣＤ显示位置的对应关系如下表 3-87 所示：

表 3-87 ＤＤＲＡＭ地址与ＬＣＤ显示位置的对应关系

	显示位置	1	2	3	4	5	6	7	……	40
DDRAM 地址	第一行	00H	01H	02H	03H	04H	05H	06H	……	27H
	第二行	40H	41H	42H	43H	44H	45H	46H	……	67H

也就是说想要在LCD1602 屏幕的第一行第一列显示一个"A"字,就要向 DDRAM 的 00H 地址写入“A”字的代码就行了。但具体的写入是要按 LCD 模块的指令格式来进行的，后面我会说到的。那么一行可有 40 个地址呀？是的，在 1602 中我们就用前 16 个就行了。第二行也一样用前 16 个地址。对应如下表 3-88：

表 3-88 1602 液晶屏显示地址

00H	01H	02H	03H	04H	05H	06H	07H	08H	09H	0AH	0BH	0CH	0DH	0EH	0FH
4H	4H	4H	4H	4H	4H	4H	4H	4H	4H	4H	4H	4H	4H	4H	4H

A 的字模如图 3-130：

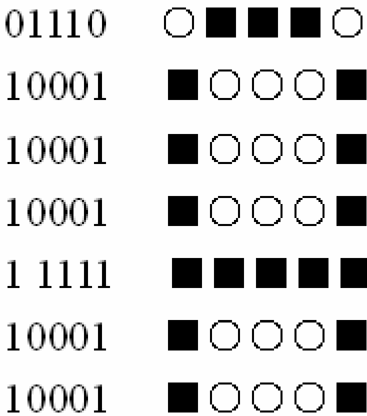


图 3-130　A 的字模

上图左边的数据就是字模数据，右边就是将左边数据用“○”代表 0，用“■”代表 1。看出是个“A”字了吗？在文本文件中“A”字的代码是 41H，PC 收到 41H 的代码后就去字模文件中将代表 A 字的这一组数据送到显卡去点亮屏幕上相应的点，你就看到“A”这个字了

RAM 地址映射图：

液晶显示模块是一个慢显示器件，所以在执行每条指令之前一定要确认模块的忙标志为低电平，表示不忙，否则此指令失效。要显示字符时要先输入显示字符地址，也就是告诉模块在，哪里显示字符，下图 3-131 是 1602 的内部显示地址。

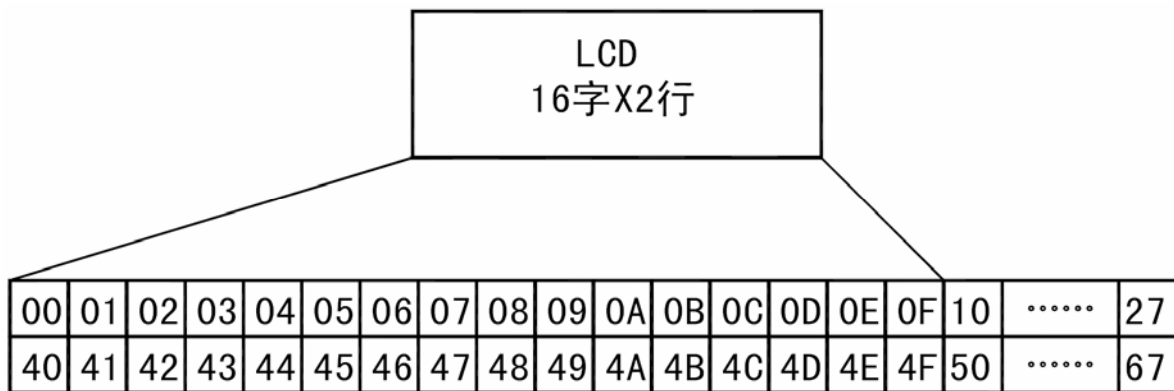


图 3-131　1602 的内部显示地址

我们知道文本文件中每一个字符都是用一个字节的代码记录的。一个汉字是用两个字节的代码记录。在 PC 上我们只要打开文本文件就能在屏幕上看到对应的字符是因为在操作系统里和 BIOS 里都固化有字符字模。什么是字模？就代表了是在点阵屏幕上点亮和熄灭的信息数据。例如“A”

例如第二行第一个字符的地址是 40H，那么是否直接写入 40H 就可以将光标定位在第二行第一个字符的位置呢？事实上我们往 DDRAM 里的 00H 地址处送一个数据，譬如 0x31(数字 A 的代码)并不能显示 A 出来。这是一个令初学者很容易出错的地方，原因就是如果你要想在 DDRAM 的 00H 地址处显示数据，则必须将 00H 加上 80H，即 80H，若要在 DDRAM 的 01H 处显示数据，则必须将 01H 加上 80H 即 81H。依次类推。大家看一下控制指令的的 8 条：DDRAM 地址的设定，即可以明白是怎么样的一回事了。

- **CGROM 和 CGRAM**

1602 液晶模块内部的字符发生存储器（CGROM)已经存储了 160 个不同的点阵字符图形，如下表所示，这些字符有：阿拉伯数字、英文字母的大小写、常用的符号、和日文假名等，每一个字符都有一个固定的代码，比如大写的英文字母“A”的代码是 01000001B（41H），显示时模块把地址 41H 中的点阵字符图形显示出来，我们就能看到字母“A”

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
xxxx0000	CG RAM (1)			0	@	P	`	p				ー	タ	ミ	α	p
xxxx0001	(2)		!	1	A	Q	a	q			。	ア	チ	ム	ä	q
xxxx0010	(3)		"	2	B	R	b	r			「	イ	ツ	メ	β	θ
xxxx0011	(4)		#	3	C	S	c	s			」	ウ	テ	モ	ε	∞
xxxx0100	(5)		$	4	D	T	d	t			、	エ	ト	ヤ	μ	Ω
xxxx0101	(6)		%	5	E	U	e	u			・	オ	ナ	ユ	σ	ü
xxxx0110	(7)		&	6	F	V	f	v			ヲ	カ	ニ	ヨ	ρ	Σ
xxxx0111	(8)		'	7	G	W	g	w			ァ	キ	ヌ	ラ	g	π
xxxx1000	(1)		(	8	H	X	h	x			ィ	ク	ネ	リ	√	x̄
xxxx1001	(2)		)	9	I	Y	i	y			ゥ	ケ	ノ	ル	⁻¹	y
xxxx1010	(3)		*	:	J	Z	j	z			ェ	コ	ハ	レ	j	千
xxxx1011	(4)		+	;	K	[	k	{			ォ	サ	ヒ	ロ	ˣ	万
xxxx1100	(5)		,	<	L	¥	l	|			ャ	シ	フ	ワ	¢	円
xxxx1101	(6)		-	=	M	]	m	}			ュ	ス	ヘ	ン	£	÷
xxxx1110	(7)		.	>	N	^	n	→			ョ	セ	ホ	゛	ñ	
xxxx1111	(8)		/	?	O	_	o	←			ッ	ソ	マ	゜	ö	█

图 3-132 CGROM 与字符的对应关系

CGRAM 是控制芯片留给用户，用以存储用户自己设计的字模编码 DDRAM 是和屏幕显示区域有对应关系的一组存储器，其功能有点中转的性质。

为了便于理解，可以举个例子： CGROM 和 CGRAM 中存储的字模信息相当于厨房中的食品，CGROM 是厨房中现成的熟食，CGRAM 是用户自行制作的菜肴，这些食品都要通过托盘转移一下，才能送到餐桌上食用；类似的字模编码都要先被读取到对应的 DDRAM 中，经如上中转以后，屏幕的相应位置才显示出字符。

2）１６０２ＬＣＤ指令说明

模块的内部操作有来自ＭＰＵ的ＲＳ、Ｒ／Ｗ、Ｅ及数据信号ＤＢ０—ＤＢ７决定。要对 DDRAM 的内容和地址进行具体操作，我们还需要根据 HD44780 的指令集浏览该指令集，并找出对 DDRAM 的内容和地址进行操作的指令。共 11 条指令，大致可以分为 4 大类：

■ 模块功能设置，如显示格式。数据长度等。

■ 设置内部ＲＡＭ地址。

■ 完成内部ＲＡＭ数据传送。

■ 完成其他功能。

一般情况下，内部ＲＡＭ的数据传送得功能使用最为频繁，因此，ＲＡＭ中的地址指针所具备的自动加一或减一功能，在一定程度上减轻了ＭＰＵ的变成。负担，此外，由于数据移位指令与写显示数据可同时进行，这样用户就能以最少的系统开发时间，达到最高的编程效率。

有一点需要特别注意：在每次访问模块之前，ＭＰＵ应首先检测忙标识ＢＦ，确认ＢＦ＝０后，访问过程才能进行。指令功能如下各表所示：

表 3-89 清屏指令

指令功能	指令编码										执行时间/ms
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	
清屏	0	0	0	0	0	0	0	0	0	1	1.64

运行时间（２５０ＫＨｚ）：１．６４ｍｓ功能：清ＤＤＲＡＭ和ＡＣ值

功能：<1> 清除液晶显示器，即将 DDRAM 的内容全部填入"空白"的 ASCII 码 20H;

<2> 光标归位，即将光标撤回液晶显示屏的左上方;

<3> 将地址计数器(AC)的值设为 0。

表 3-90 光标归位指令

指令功能	指令编码										执行时间/ms
	RS	R/W	DB7	DB6	DB5	DB4	D3	DB2	DB1	DB0	
光标归为	0	0	0	0	0	0	0	0	1	X	1.64

运行时间（２５０ＫＨｚ）：１．６４ｍｓ

功能：ＡＣ＝０，光标、画面回ＨＯＭＥ位

<1> 把光标撤回到显示器的左上方;

<2> 把地址计数器(AC)的值设置为 0;

<3> 保持 DDRAM 的内容不变

表 3-91 进入模式设置指令

指令功能	指令编码										执行时间/us
	RS	R/W	DB7	DB6	DB5	DB4	D3	D2	DB1	DB0	
进入模式设置	0	0	0	0	0	0	0	1	I/D	S	40

运行时间（２５０ＫＨｚ）：４０μｓ功能：设置光标、画面移动方式

设定每次定入 1 位数据后光标的移位方向，并且设定每次写入的一个字符是否移动。参数设定的情况如下所示：

位名　　设置

I/D　　0=写入新数据后光标左移

　　　　1=写入新数据后光标右移

S　　　0=写入新数据后显示屏不移动

　　　　1=写入新数据后显示屏整体右移 1 个字

表 3-92 显示开关控制指令

指令功能	指令编码										执行时间/us
	RS	R/W	DB7	DB6	DB5	DB4	D3	D2	D1	DB0	
显示开关控制	0	0	0	0	0	0	1	D	C	B	40

运行时间（２５０ＫＨｚ）：４０μｓ

功能：设置显示、光标及闪烁开、关

参数设定的情况如下：

位名	设置	
D	0=显示功能关	1=显示功能开
C	0=无光标	1=有光标
B	0=光标闪烁	1=光标不闪烁

表 3-93 设定显示屏或光标移动方向指令

指令功能	指令编码										执行时间/us
	RS	R/W	DB7	DB6	DB5	DB4	D3	D2	DB1	DB0	
设定显示屏或光标移动方向	0	0	0	0	0	1	S/C	R/L	X	X	40

运行时间（２５０ＫＨｚ）：４０μｓ

功能：光标、画面移动，不影响ＤＤＲＡＭ

使光标移位或使整个显示屏幕移位。参数设定的情况如下：

S/C	R/L	设定情况
0	0	光标左移 1 格，且 AC 值减 1
0	1	光标右移 1 格，且 AC 值加 1
1	0	显示器上字符全部左移一格，但光标不动
1	1	显示器上字符全部右移一格，但光标不动

表 3-94 功能设定指令

指令功能	指令编码										执行时间/us
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	
功能设定	0	0	0	0	1	DL	N	F	X	X	40

运行时间（２５０ＫＨｚ）：４０μｓ

功能：工作方式设置（初始化指令）

设定数据总线位数、显示的行数及字型。参数设定的情况如下：

位名	设置
DL	0=数据总线为 4 位
	1=数据总线为 8 位
N	0=显示 1 行
	1=显示 2 行

F　　　　0=5×7 点阵/每字符
　　　　　1=5×10 点阵/每字符

表 3-95 设定 CGRAM 地址指令

指令功能	指令编码										执行时间/us
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	
设定 CGRAM 地址	0	0	0	1	CGRAM 的地址（6 位）						40

运行时间（２５０ＫＨｚ）：４０μｓ
功能：设置ＣＧＲＡＭ地址。Ａ５—Ａ０＝０—３ＦＨ
设定下一个要存入数据的 CGRAM 的地址。

表 3-96 设定 DDRAM 地址指令

指令功能	指令编码										执行时间/us
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	
设定 DDRAM 地址	0	0	1	DDRAM 的地址（7 位）							40

运行时间（２５０ＫＨｚ）：４０μｓ
功能：设置ＤＤＲＡＭ地址
说明：

- ■ Ｎ＝０，一行显示Ａ６—Ａ０＝０—４ＦＨ。
- ■ Ｎ＝１，两行显示，首行Ａ６—Ａ０＝００Ｈ—２ＦＨ，次行Ａ６—Ａ0。

(注意这里我们送地址的时候应该是 0x80+Address，这也是前面说到写地址命令的时候要加上 0x80 的原因)

表 3-97 读取忙信号或 AC 地址指令

指令功能	指令编码										执行时间/us
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	
读取忙碌信号或 AC 地址	0	0	F8	AC 内容（7 位）							40

功能：读忙ＢＦ值和地址计数器ＡＣ值
说明：ＢＦ＝１：忙　；　ＢＦ＝０：准备好。读取忙碌信号 BF 的内容，BF=1 表示液晶显示器忙，暂时无法接收单片机送来的数据或指令；当 BF=0 时，液晶显示器可以接收单片机送来的数据或指令
此时，ＡＣ值即为最近一次地址设置（ＣＧＲＡＭ或ＤＤＲＡＭ）定义。

表 3-98 数据写入 DDRAM 或 CGRAM 指令一览

指令功能	指令编码										执行时间/us
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	
数据写入到 DDRAM	1	0	要写入的数据 D7——D0								40

或 CGRAM				

运行时间：（２５０ＫＨｚ）：４０μｓ

功能：<1> 将字符码写入 DDRAM，以使液晶显示屏显示出相对应的字符;

<2> 将使用者自己设计的图形存入 CGRAM。

表 3-99 从 CGRAM 或 DDRAM 读出数据的指令一览

指令功能	指令编码										执行时间/us
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	
从 CGRAM 或 DDRAM 读出数据	1	1	要写入的数据 D7——D0								40

运行时间：（２５０ＫＨｚ）：４０μｓ

功能：读取 DDRAM 或 CGRAM 中的内容。

基本操作时序：

读状态　　输入：RS=L，RW=H，E=H

　　　　　输出：DB0～DB7=状态字

写指令　　输入：RS=L，RW=L，E=下降沿脉冲，DB0～DB7=指令码

　　　　　输出：无

读数据　　输入：RS=H，RW=H，E=H

　　　　　输出：DB0～DB7=数据

写数据　　输入：RS=H，RW=L，E=下降沿脉冲，DB0～DB7=数据

　　　　　输出：无

3）如何显示一个自定义的字符

步骤如下：

1. 先将自定义字符写入 CGRAM

2. 再将 CGRAM 中的自定义字符送到 DDRAM 中显示

我们从 CGROM 表上可以看到，在表的最左边是一列可以允许用户自定义的 CGRAM，从上往下看着是 16 个，实际只有 8 个字节可用。它的字符码是 00000000—00000111 这 8 个地址，表的下面还有 8 个字节，但因为这个 CGRAM 的字符码规定 0—2 位为地址，3 位无效，4—7 全为 0。因此 CGRAM 的字符码只有最后三位能用也就是 8 个字节了。等效为 0000X111，X 为无效位，最后三位为 000—111 共 8 个。如果我们想显示这 8 个用户自定义的字符，操作方法和显示 CGROM 的一样，先设置 DDRAM 位置，再向 DDRAM 写入字符码，例如“A”就是 41H。现在我们要显示 CGRAM 的第一个自定义字符，就向 DDRAM 写入 00000000B（00H），如果要显示第 8 个就写入 00000111（08H）。

我们来看下怎么向这 8 个自定义字符写入字模。如下表 3-100 为设置 CGRAM 地址

表 3-100 设置 CGRAM 地址的指令

指令功能	指令编码										执行时间/us
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	
设定 CGRAM 地址	0	0	0	1	CGRAM 的地址（6 位）						40

从这个指令可以看出指令数据的高 2 位已固定是 01，只有后面的 6 位是地址数据，而这 6 位中的高 3 位就表示这 8 个自定义字符，最后的 3 位就是字模数据的八个地址了。例如

第一个自定义字符的字模地址为 01000000—01000111 八个地址。我们向这 8 个字节写入字模数据，让它能显示出“C”，如图 3-133：



图 3-133 “C”字模数据

写入时先设置 CGRAM 地址 0x40；显示是直接取 CGRAM 的数据

3.20.4 硬件连接原理

神舟51开发板板载有一个1602液晶屏模块接口连接器。单片机的IO与1602液晶屏模块相连。如图3-134所示：

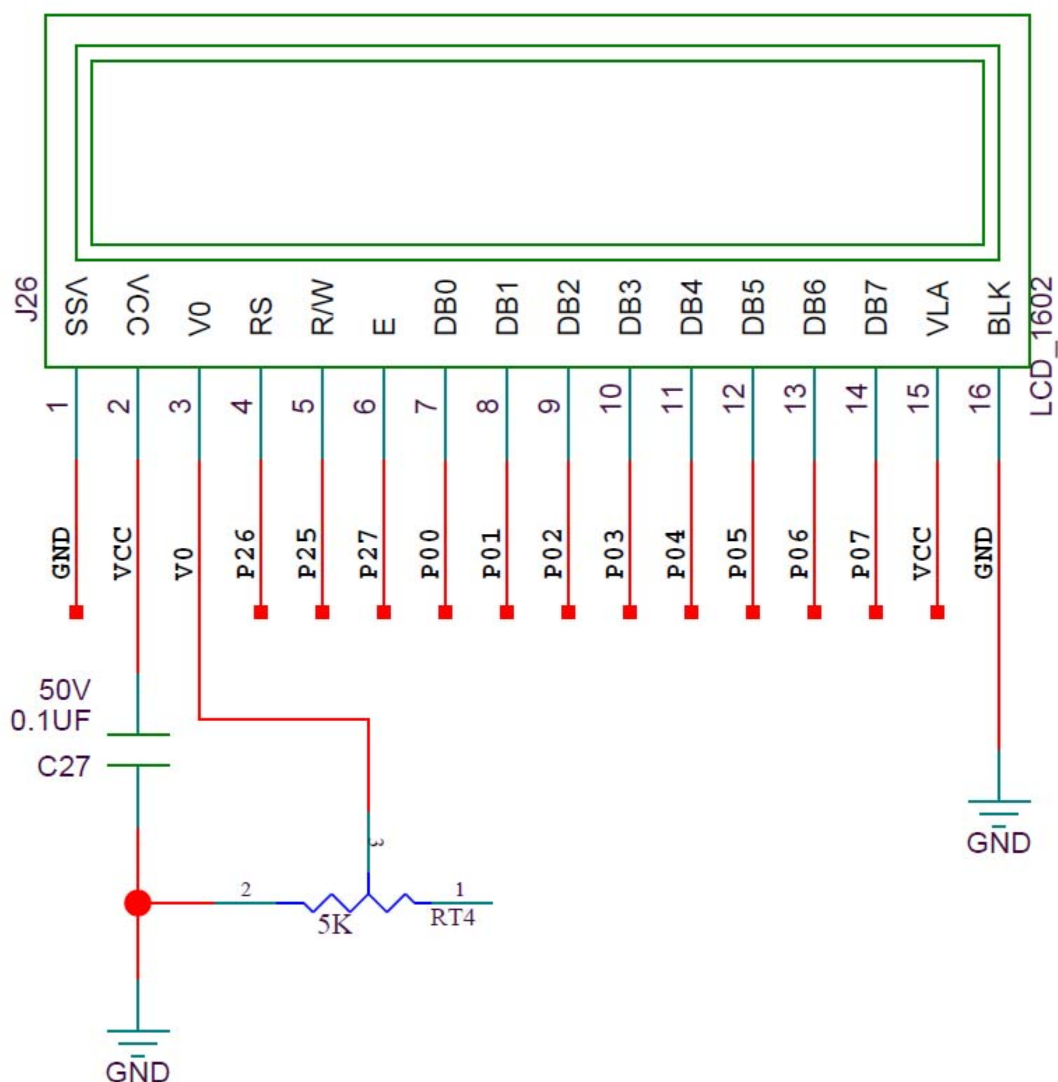


图3-1341602液晶屏硬件连接原理图

其中 1602 的 V0 是 1602 液晶的对比度调节管脚,如果液晶不显示或者显示文字不清晰,可以调节 RT4 电位器，来达到最理想的显示效果。

3.20.5 例程 01 LCD1602 静态显示实验

代码如下：

```
/***********************************************************************
* 例程：LCD1602静态显示
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：单片机控制LCD1602液晶模块静态显示ASCII字符
* 现象：通过本例程了解LCD1602液晶模块的基本原理，理解并掌握通过单片机驱动
*       LCD1602液晶模块的方法。如果看到LCD1602液晶模块第一行静态显示字符串
*       "OK"，第二行显示"www.ARMJISHU.com"，表明LCD1602液晶模块显示正常
*引脚定义如下：1-VSS 2-VDD 3-V0 4-RS 5-R/W 6-E 7-14 DB0-DB7 15-BLA 16-BLK
```

```
*****************************************************************************/
//包含头文件，一般情况不需要改动，头文件包含特殊功能寄存器的定义
#include<reg52.h>
#include<intrins.h>
//定义端口
sbit RS = P2^6;
//RS为寄存器选择，高电平时选择数据寄存器、低电平时选择指令寄存器。
sbit RW = P2^5;     //R/W为读写信号线，高电平时进行读操作，低电平时进行写操作。
sbit EN = P2^7;     //E端为使能端，当E端由高电平跳变成低电平时，液晶模块执行命令。
#define RS_CLR RS=0
#define RS_SET RS=1
#define RW_CLR RW=0
#define RW_SET RW=1
#define EN_CLR EN=0
#define EN_SET EN=1
#define DataPort P0
/*------------------------------------------------------
 uS延时函数
 含有输入参数 unsigned char t，无返回值
 unsigned char 是定义无符号字符变量，其值的范围是
 0~255 这里使用晶振12M，精确延时请使用汇编,大致延时
 长度如下 T=tx2+5 uS
------------------------------------------------------*/
void DelayUs2x(unsigned char t)
{
    while(--t);
}
/*------------------------------------------------------
 mS延时函数
 含有输入参数 unsigned char t，无返回值
 unsigned char 是定义无符号字符变量，其值的范围是
 0~255 这里使用晶振12M，精确延时请使用汇编
------------------------------------------------------*/
void DelayMs(unsigned char t)
{
    while(t--)
    {
        //大致延时1mS
        DelayUs2x(245);
        DelayUs2x(245);
    }
}
/*------------------------------------------------------
              判忙函数
```

```
------------------------------------------------------*/
bit LCD_Check_Busy(void)
{
    DataPort= 0xFF;
    //当RS为低电平R/W为高电平时可以读忙信号
    RS_CLR;
    RW_SET;
    EN_CLR;
    _nop_();
    EN_SET;
    return (bit)(DataPort & 0x80);
}
/*------------------------------------------------------
                    写入命令函数
------------------------------------------------------*/
void LCD_Write_Com(unsigned char com)
{
    while(LCD_Check_Busy());       //忙则等待
    //当RS和R/W共同为低电平时可以写入指令或者显示地址
    RS_CLR;
    RW_CLR;
    EN_SET;
    DataPort= com;
    _nop_();
    EN_CLR;
}
/*------------------------------------------------------
                     写入数据函数
------------------------------------------------------*/
void LCD_Write_Data(unsigned char Data)
{
    while(LCD_Check_Busy());     //忙则等待
    //当RS为高电平R/W为低电平时可以写入数据
    RS_SET;
    RW_CLR;
    EN_SET;
    DataPort= Data;
    _nop_();
    EN_CLR;
}
/*------------------------------------------------------
                清屏函数
------------------------------------------------------*/
void LCD_Clear(void)
```

```
{
    LCD_Write_Com(0x01);
    DelayMs(5);
}

/*------------------------------------------------------
                写入字符串函数
------------------------------------------------------*/
void LCD_Write_String(unsigned char x,unsigned char y,unsigned char *s)
{
    if (y == 0)
    {
        LCD_Write_Com(0x80 + x);        //表示第一行
    }
    else
    {
        LCD_Write_Com(0xC0 + x);        //表示第二行
    }

    while (*s)
    {
        LCD_Write_Data( *s);
        s ++;
    }
}
/*------------------------------------------------------
              写入字符函数
------------------------------------------------------*/
void LCD_Write_Char(unsigned char x,unsigned char y,unsigned char Data)
{
    if (y == 0)
    {
        LCD_Write_Com(0x80 + x);
    }
    else
    {
        LCD_Write_Com(0xC0 + x);
    }
    LCD_Write_Data( Data);
}
/*------------------------------------------------------
                       初始化函数
------------------------------------------------------*/
void LCD_Init(void)
```

```
{
      LCD_Write_Com(0x38);       /*显示模式设置*/
      DelayMs(5);
      LCD_Write_Com(0x38);
      DelayMs(5);
      LCD_Write_Com(0x38);
      DelayMs(5);
      LCD_Write_Com(0x38);
      LCD_Write_Com(0x08);       /*显示关闭*/
      LCD_Write_Com(0x01);       /*显示清屏*/
      LCD_Write_Com(0x06);       /*显示光标移动设置*/
      DelayMs(5);
      LCD_Write_Com(0x0C);       /*显示开及光标设置*/
}

/*------------------------------------------------------
                    主函数
------------------------------------------------------*/
void main(void)
{
      LCD_Init();
      LCD_Clear();//清屏
      while (1)
      {
            //测试单个字符写入显示
            LCD_Write_Char(7,0,'O');
            LCD_Write_Char(8,0,'K');
            //测试字符串写入显示
            LCD_Write_String(0,1,"www.ARMJISHU.com");
            while(1);
      }
}
```

本实验单片机控制 LCD1602 液晶模块静态显示 ASCII 字符。如果看到 LCD1602 液晶模块第一行静态显示字符串"OK"，第二行显示"www.ARMJISHU.com"，表明 LCD1602 液晶模块显示正常，否则实验失败。

本实验无需任何额外的线缆连接。

实验效果图如下图3-135所示：

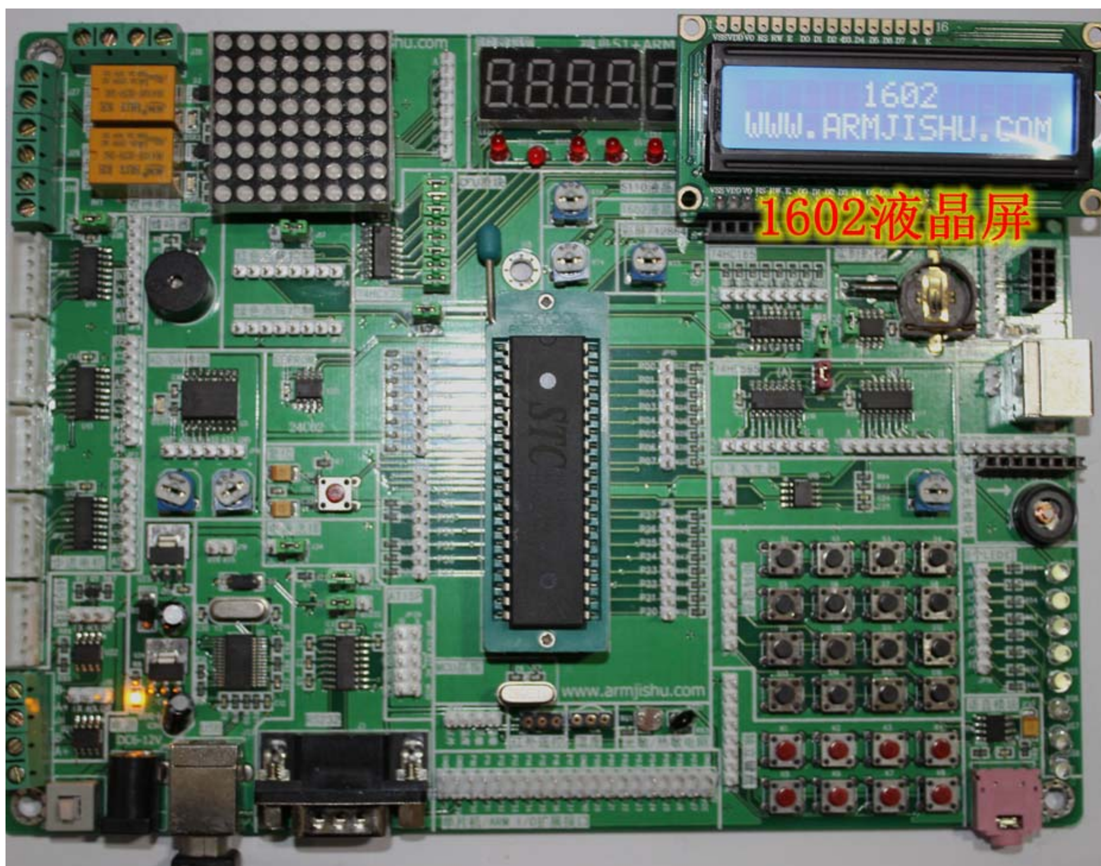


图 3-135 硬件连接实物图

3.20.6 更多有关 1602 液晶屏的例程

更多 1602 液晶屏相应的例程可通过北京航空航天大学出版社下载专区下载，如下表 3-101：

表 3-101 1602 液晶屏更多丰富例程（含详细注释和文档分析）

序号	例程功能
例程 01	LCD1602 静态显示实验
例程 02	LCD1602 闪烁显示实验
例程 03	LCD1602 动态显示字符
例程 04	LCD1602 液晶滚动显示
例程 05	LCD1602 光标移动显示
例程 06	LCD1602 显示按键输入
例程 07	串口中断方式接收数据在 1602 液晶屏显示

3.21 红外遥控器收发

3.21.1 红外收发的简介

在了解红外收发前，我们先来了解一下什么是红外，在日常生活中，我们通过眼睛能看到各种颜色的光。那我们是否了解各种颜色的光之间有什么区别的呢？难道就只是颜色不一样？那为什么光又分为那么多种颜色呢？

其实，光的颜色是由它的波长来决定的，那么波长又是什么呢？为了能让用户更能理解它的定义，我们不是科学家，不深入研究，只是了解一下它的定义就行了，我们前面知道了知道光的颜色是由波长决定的。光的波长就是光在单位时间内通过的位移（位移是从空间的一个位置运动到另一个位置，它的位置变化叫做在这一运动过程中的位移）。光的本质是电磁波，所以就会有频率的产生。光的频率在传播中保持不变，意思是在光通过不同介质的时候，频率不变而波长发生改变。因此，光的波长由光的频率(颜色)，以及传播的介质决定。

波长的单位我们可以用 nm 或者 um 来表示，各种颜色有各自的波长，人的眼睛能看到的可见光按波长从长到短排列，依次为红、橙、黄、绿、青、蓝、紫。其中红光的波长范围为 620～760nm；紫光的波长范围为 380～440nm。比紫光波长还短的光叫紫外线，比红光波长还长的光叫红外线。在 1800 年 4 月 24 日英国伦敦皇家学会（ROYAL SOCIETY）的威廉·赫歇尔发表太阳光在可见光谱的红光之外还有一种不可见的延伸光谱，具有热效应。他所使用的方法很简单，用一支温度计测量经过棱镜分光后的各色光线温度，由紫到红，发现温度逐渐增加，可是当温度计放到红光以外的部分，温度仍持续上升，因而断定有红外线的存在。红外线遥控就是利用波长为 760nm～400um 之间的近红外线来传送控制信号的。下图 3-136 为各种光的波长示意图，红光外的为红外光，紫色光以外的为紫外光。

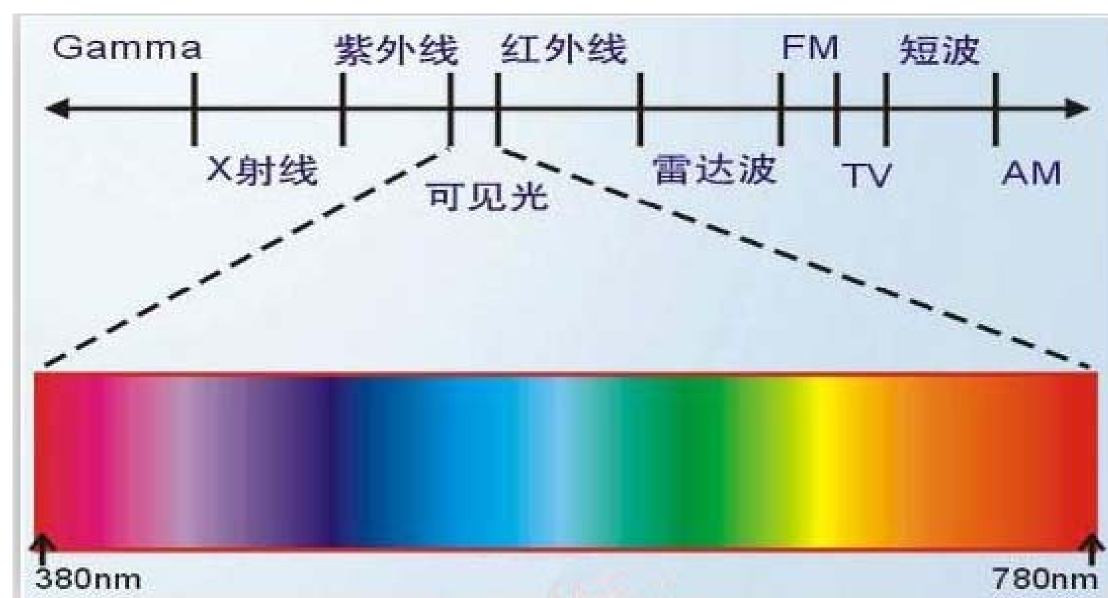


图 3-136 各种光的波长示意图

3.21.2 红外收发的特点与用途

1．红外的特点有：

- 具有很强热效应
- 易于被物体吸收
- 穿透能力比可见光强
- 小角度（30 度锥角以内），短距离
- 点对点直线数据传输，保密性强
- 传输速率较高

2．红外的用途有：

通讯技术———常被应用在计算机及其外围设备、移动电话、数码相机、工业设备、网络接入设备，如调制解调器等，像我们这章所用到的红外收发，还有深海探测等无线通讯方面的技术

医疗用途———红外线对人体皮肤、皮下组织具有强烈的穿透力。外界红外线辐射人体产生的一次效应可以使皮肤和皮下组织的温度相应增高，促进血液的循环和新陈代谢，促进人的健康，红外线还有杀菌的能力。

因为我们这章用到的是红外通讯技术，所以它的其他用途我们就不作说明了，主要说下用作红外收发通讯的用途。

红外通讯就是通过红外线传输数据。在电脑技术发展早期，数据都是通过线缆传输的，线缆传输连线麻烦，需要特制接口，颇为不便。无线通信技术逐渐发展起来。

3.21.3 红外的发送工作原理

红外遥控有发送和接收两个组成部分。发送端将待发送的二进制信号编码调制为一系列的脉冲串信号，通过红外发射管发射红外信号。红外接收完成对红外信号的接收、放大、检波、整形，并解调出遥控编码脉冲，流程图如下图 3-137：

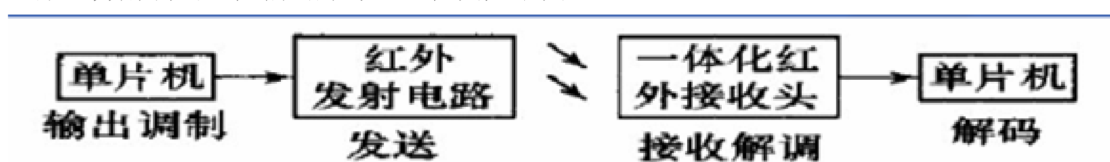


图 3-137 红外通信流程图

把需要发送的信号编码调制为一系列的脉冲串信号由红外 LED 发送出去。

二进制信号的调制由单片机来完成，它把编码后的二进制信号调制成一定频率的间断脉冲串，相当于用二进制信号的编码乘以这个间断脉冲信号得到的间断脉冲串，即是调制后用于红外发射二极管发送的信号，如下图 3-138 所示，要使红外发光二极管产生调制光，只需在它的驱动管上加上一定频率的脉冲电压。一般由键盘电路、红外编码芯片、电源和红外发射电路组成

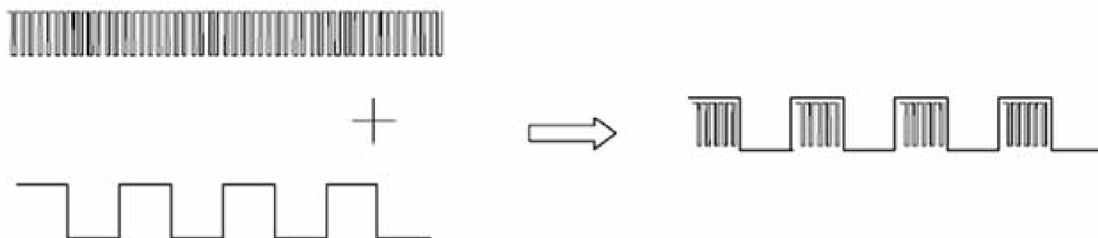


图 3-138 二进制码的调制

在同一个遥控电路中通常要使用实现不同的遥控功能或区分不同的机器类型，这样就要求信号按一定的编码传送，编码则会由编码芯片或电路完成。通过对用户码的检验，每个遥控器只能控制一个设备动作，这样可以有效地防止多个设备之间的干扰。就像是电视机、空调有专门的遥控器，遥控器上的按键各有各自的功能。

红外遥控发射芯片采用 PPM 编码方式（脉冲位置调制,又称脉位调制，由头码+脉冲数组成,称为 PPM 编码方式），当发射器按键按下后，将发射一组 108ms 的编码脉冲。遥控编码脉冲由前导码、16 位地址码（8 位地址码、 8 位地址码的反码）和 16 位操作码（8 位操作码、 8 位操作码的反码）组成, 反码表示正数的反码与其原码相同；负数的反码是对其原码逐位取反，但符号位除外。由一个 9ms 的高电平 (起始码) 和一个 4. 5ms 的低电平 (结果码) 组成作为接受数据的准备脉冲。解码的关键是如何识别“0”和“1”，从位的定义我们可以发现“0”、“1”均以以脉宽为 0. 56ms 的电平开始、周期为 1. 12ms 的组合表示二进制的“0”；以脉宽为 1. 68ms 、周期为 2. 24ms 的组合表示二进制的“1”，根据这样的定义，我们就能读出发送的数据是什么了。如下图 3-139 所示：

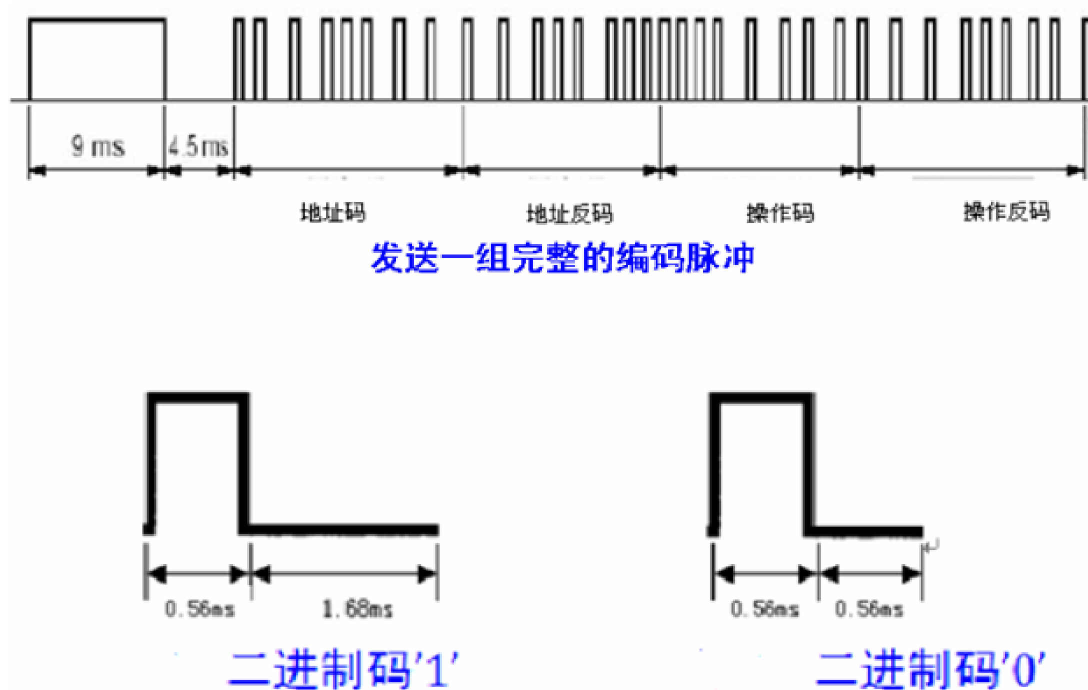


图 3-139 发送的数据

下图 3-140、3-141 为我们的红外发送装置，它由键盘电路、红外编码芯片、电源和红外发射电路组成，组合成一起就能把按键编码成一定的数据发送出去给红外接收了



图 3-140 红外发送装置（产生信号）



图 3-141 红外发送装置（发送信号）

3.21.4 红外的接收头的物理结构工作原理

我们首先了解一下红外接收头，再看红外接收原理：

红外线接收头是目前使用非常广泛的一种通信和遥控器件，在现实生活中几乎随处可见，例如电视机、录像机、空调机等等，都"不约而同"地采用红外线遥控，它的广泛使用源于它多方面的优点：抗干扰能力好、编码及解码容易、功耗小、成本低等。我们可以从它的特性、实物图与尺寸及引脚定义和它的接收方式去了解它。

红外接收头的特性：

●小型设计；

●内置专用 IC；

●宽角度及长距离接收；

●抗干挠能力强；

●能抵挡环境干挠光线；

●低电压工作；

红外接收头实物图如下图 3-142 所示：



图 3-142 红外接收头实物图

尺寸及引脚定义结构图如下图 **3-143**、**3-144** 所示：

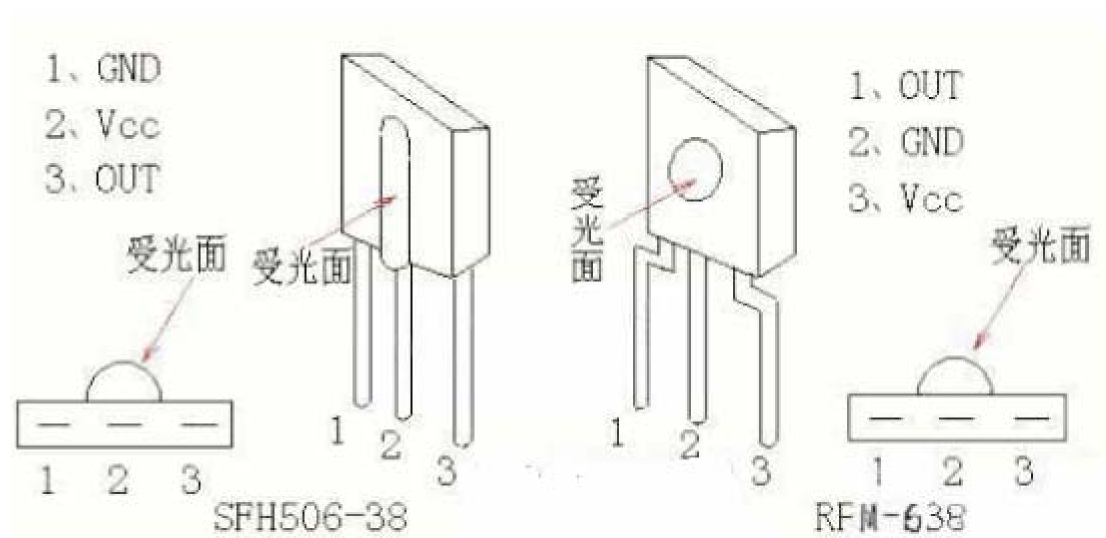


图 3-143 引脚定义结构图

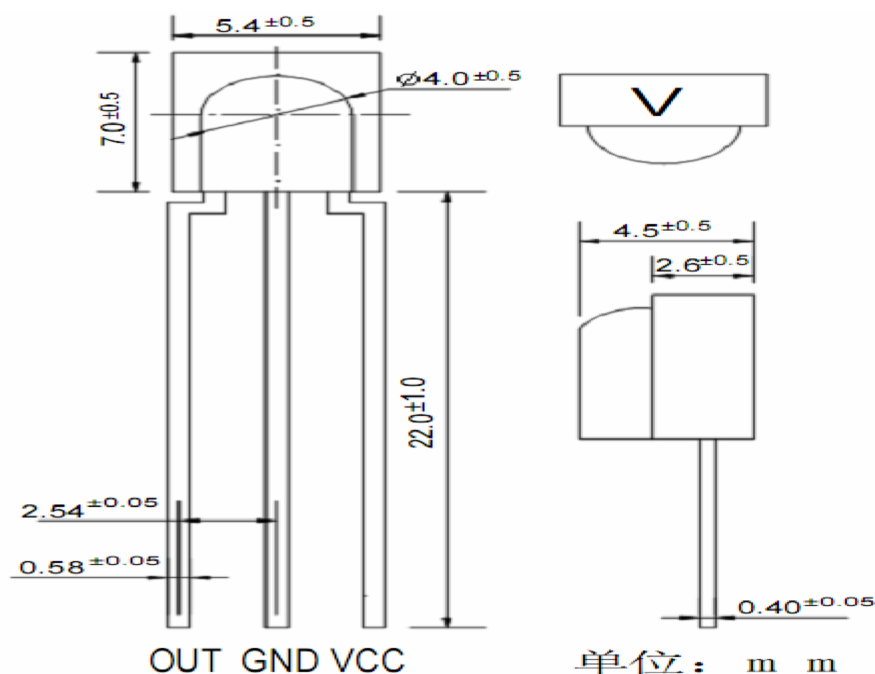


图 3-144 尺寸图

接收角度如下图 3-145 所示：

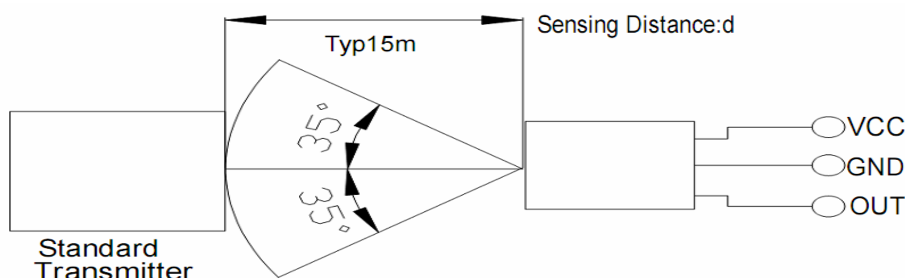


图 3-145 接收角度

3.21.5 红外的接收头的工作原理

了解了红外接收头后，我们再来了解下它是如何工作的，它的原理是什么。红外接收头内置接收管将红外发射管发射出来的光信号转换为微弱的电信号，此信号经由 IC 内部放大器进行放大，然后通过接收管的处理将波形整形后还原为遥控器发射出的原始编码，经由接收头的信号输出脚输入到电器上的编码识别电路，红外解码的关键就是识别 0 和 1。

我们前面介绍过红外接收是完成对红外信号的接收、放大、检波、整形，并解调出遥控编码脉冲的，这个是由接收头内部完成的。经过它的接收放大和解调会在输出端直接输出原始的信号

红外接收头一般是接收、放大、解调一体头，一般红外信号经接收头解调后，数据“0”和“1”的区别通常体现在高低电平的时间长短或信号周期上，这个我们上面有介绍。单片机解码时，通常将接收头输出脚连接到单片机的外部中断，结合定时器判断外部中断间隔的时间从而获取数据。重点是找到数据“0”与“1”间的波形差别。

在这里特别强调：编码与解码是一对逆过程，不仅在原理上是一对逆过程，在码的发收

过程也是互反的，即以前发射端原始信号是高电平,那接收头输出的就是低电平，反之亦然。因此为了保证解码过程简单方便，在编码时应该直接换算成其反码。如下图 3-146 为红外发送与红外接收头接收的电平对比情况

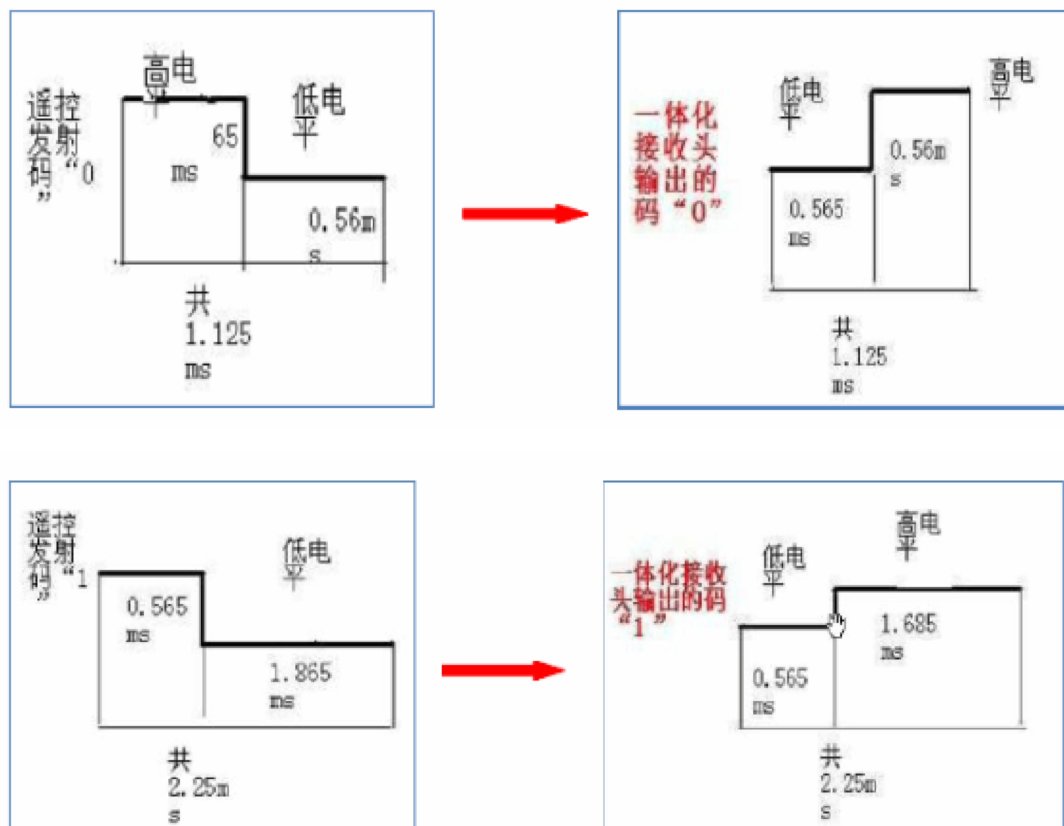


图 3-146 红外发送与红外接收头接收的电平对比情况

3.21.6 红外的接收的过程描述

接收电路的红外接收管是一种光敏二极管，使用时要给红外接收二极管加反向偏压，它才能正常工作而获得高的灵敏度。红外接收二极管一般有圆形和方形两种。由于红外发光二极管的发射功率较小，红外接收二极管收到的信号较弱，所以接收端就要增加高增益放大电路。然而现在不论是业余制作或正式的产品，大都采用成品的一体化接收头，红外线一体化接收头是集红外接收、放大、滤波和比较器输出等的模块，性能稳定、可靠。所以，有了一体化接收头，人们不再制作接收放大电路，这样红外接收电路不仅简单而且可靠性大大提高。红外接收头包含两个芯片，一个是 PD（即红外接收管）,一个是 IC。其中 PD 接收来自发射管的光信号（该信号已被调制），将光信号转换为电信号，即光电转换，常用于光接收器中。PD 芯片属于典型的 PIN 结构光电二极管。由 PD 接收转换而来的电信号通过 IC 进行放大，自动增益控制，滤波，解调，波形整形，比较器输出交由后面的电路进行识别还原。以上就是红外接收头的接收过程。

红外接收头在完成解码的时候，通过接收头的信号输出脚输入到电器上的编码识别电路或者是单片机中完成各种操作，如我们例程中的控制 LED、在数码管或者是 1602 屏上显示接收到的数据。

3.21.7 硬件原理图与连接

硬件连接原理图如下图 3-147 所示：

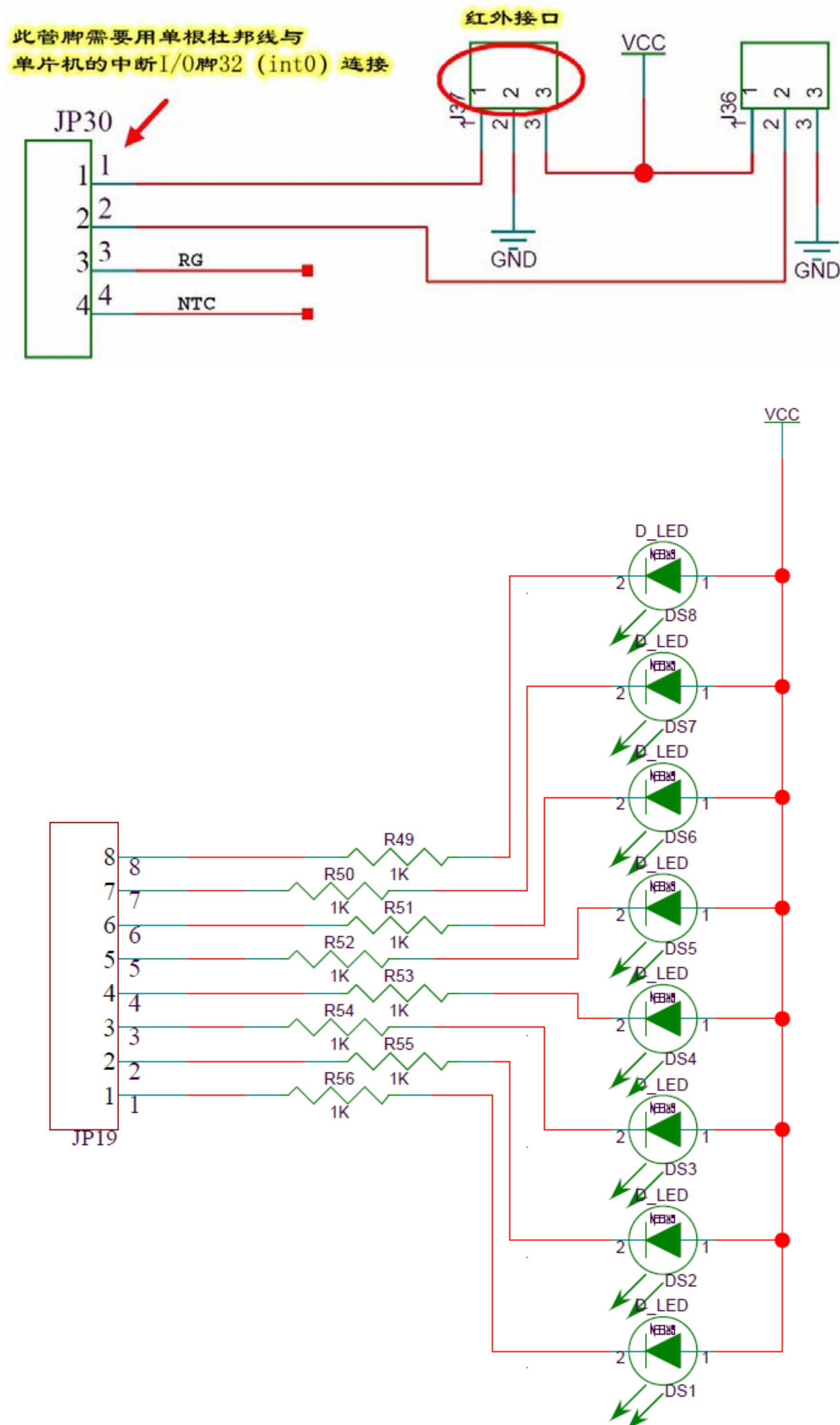


图 3-147 硬件原理图

J37 插入红外接收头后，板子上的 JP30 插针的管脚 1 通过一根单针杜邦线与 JP14 的 P32 管脚连接；再用另一根单针杜邦线将 JP16 的 P20 管脚与 JP19 插针的管脚 1 连接，如下图 3-148：

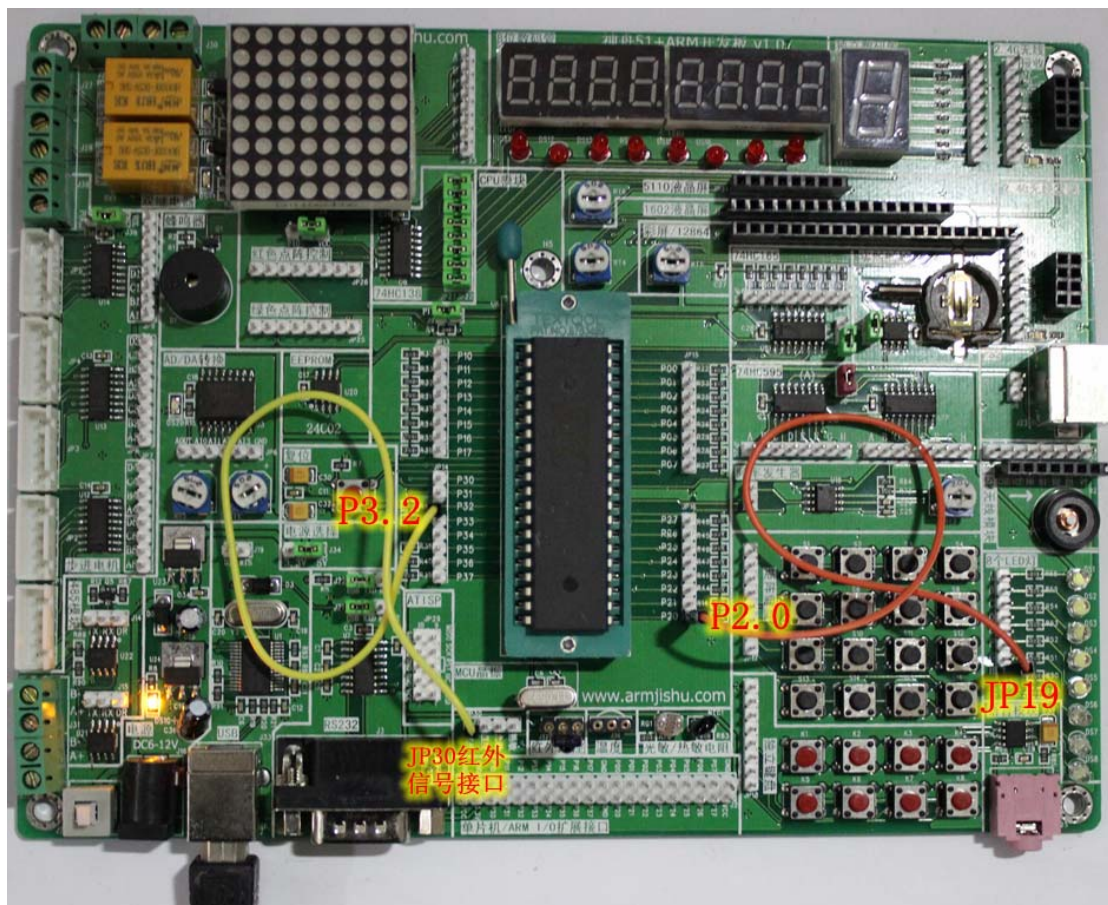


图 3-148 硬件环境连接实物图

3.21.8 例程 01 红外控制LED灯闪烁

代码如下：

```
/*******************************************************
* 例程：红外接收原理
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：从红外接收头接收到信息反映到 LED 上
*******************************************************/
#include<reg52.h> //包含头文件，头文件包含特殊功能寄存器的定义
sbit LED=P2^0;// 用 sbit 关键字 定义 LED 到 P2.0 端口
sbit IR_IN=P3^2;
/*-----------------------------------------------
                主函数
-----------------------------------------------*/
void main (void)
```

```
{
    while (1)          //主循环(while()  是单片机的一种基本循环模式。
                       //当满足条件时进入循环，不满足跳出)
    {
        LED=IR_IN;     //主循环中添加其他需要一直工作的程序
    }
}
```

硬件连接关系如下表 3-102 所示：

表 3-102 硬件连接对应关系表

单片机接口	插座 1	方式	插座 2	线缆	功能
P20 脚	JP16	直连	JP19 的管脚 1	1 根单针杜邦电缆	控制单个 LED 灯
P32 脚	JP14	直连	JP30 的管脚 1	1 根单针杜邦电缆	单片机接收红外接收头送出的信号
注意：将红外接收头插入 J37（注意弧度方向）					
实验现象：下载程序后，使用我们提供的遥控器，按下按键，LED 灯会不规则的闪烁					

知识要点：

我们使用遥控器按下键值后，红外接收头会将相应的键值转换成高低电平的数字信号输出给单片机，单片机再控制 LED 灯做出不规则的闪烁。

3.21.9 更多有关红外遥控器的例程

更多红外收发遥控器相应的例程可通过北京航空航天大学出版社下载专区下载，如下表 3-103：

表 3-103 红外收发更多丰富例程（含详细注释和文档分析）

序号	例程功能
例程 01	红外控制 LED 灯闪烁
例程 02	红外解码数码管显示
例程 03	红外解码 1602 液晶屏显示

3.22 热敏/光敏电阻

3.22.1 为什么会有热敏/光敏电阻出现?

能随温度变化，阻值对应发生改变的电阻我们称它为热敏电阻，能随亮度变化，阻值对应发生改变的电阻我们称它为光敏电阻。

为什么会出现热敏电阻呢？当煮开水的时候人不在，而水开了怎么办呢？发生火灾的时候没人看到怎么办呢？还有就是一些机器运行过热了，有危险而又没人知道又会怎么样，为解决这些问题，我们生产出了热敏电阻这个期间，上面提到的问题它都能帮我们解决，当温度发生改变的时候，它的阻值也会随即改变，达到一定温度的时候就能提醒用户，如水开了

温度上升，它的阻值下降到一定的时候，启动一些电路，如亮灯或者是响喇叭等，火灾、机器过载也是同样的道理。

那为什么又会出现光敏电阻呢？路灯在天黑的时候我们想自动点亮路灯而不需要手动去控制、需要一个自动调节亮度的装置，亮度降低时，调节亮度增加，达到一个稳定的效果。我们需要一个自动检测并执行的器件，于是光敏电阻诞生了，这些问题都能通过光敏电阻去实现了。

3.22.2 热敏电阻的工作和制造原理

我们前面了解到了热敏电阻，实物图如下图 3-149 所示，知道它是通过温度改变阻值的，那它的阻值都有什么变化呢？我们大致可以分为 2 种，一种是温度升高，阻值降低，这个是大多数热敏电阻特性，另外小部分是温度升高，阻值相对也增加的，除了这个特性外，热敏电阻还有个特点是灵敏度，灵敏度就是温度变化了，热敏电阻经过多长的时间就可以感知了；温度变化多少，它的阻值才会发生变化，灵敏度越高的热敏电阻在温度发生变化时就更快感知，阻值也会相对应的发生变化。这里还要说一下的是它的工作温度范围，常温器件适用于-55℃～315℃，高温器件适用温度高于 315℃（目前最高可达到 2000℃），低温器件适用于-273℃～55℃；这个是由它的制造材料所决定的。



图 3-149 热敏电阻

基于内光电效应。在半导体光敏材料两端装上电极引线，将其封装在带有透明窗的管壳里就构成光敏电阻，为了增加灵敏度，两电极常做成梳状。用于制造光敏电阻的材料主要是金属的硫化物、硒化物和碲化物等半导体。通常采用涂敷、喷涂、烧结等方法在绝缘衬底上制作很薄的光敏电阻体及梳状欧姆电极，接出引线，封装在具有透光镜的密封壳体内，以免受潮影响其灵敏度。入射光消失后，由光子激发产生的电子—空穴对将复合，光敏电阻的阻值也就恢复原值。在光敏电阻两端的金属电极加上电压，其中便有电流通过，受到一定波长的光线照射时，电流就会随光强的增大而变大，从而实现光电转换。光敏电阻没有极性，纯粹是一个电阻器件，使用时既可加直流电压，也加交流电压。半导体的导电能力取决于半导体导带内载流子数目的多少。

3.22.3 光敏电阻的工作和制造原理

我们前面也了解到了光敏电阻，实物图如下图 3-150 所示，知道它是通过光的亮度改变阻值的，那它的阻值都有什么变化呢？当入射光强时，电阻减小，入射光弱，电阻增大。还

有另一种入射光弱，电阻减小，入射光强，电阻增大。只要人眼可感受的光，都会引起它的阻值变化。和热敏电阻一样，它也有一个灵敏度、响应时间。光的亮度变化了，光敏电阻经过多长的时间就可以感知了；亮度变化多少，它的阻值才会发生变化，灵敏度越高的光敏电阻在光的亮度发生变化时就更快感知，阻值也会相对应的发生变化。但它的响应速度不快，在 ms 到 s 之间。

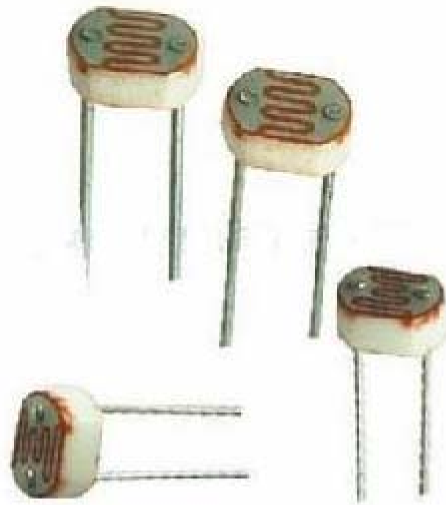


图 3-150 光敏电阻

热敏电阻用到了一种叫 PTC 的材料，PTC 即正温度系数效应，仅指此材料的电阻会随温度的升高而增加。如大多数金属材料都具有 PTC 效应。在这些材料中，PTC 效应表现为电阻随温度增加而线性增加，这就是通常所说的线性 PTC 效应。

当电路正常工作时，热敏电阻温度与室温相近、电阻很小，串联在电路中不会阻碍电流通过；而当电路因故障而出现过电流时，热敏电阻由于发热功率增加导致温度上升，当温度超过开关温度时，电阻瞬间会剧增，回路中的电流迅速减小到安全值。热敏电阻动作后，电路中电流有了大幅度的降低

3.22.4 硬件电路原理图

硬件连接原理图如下图 3-151 所示：

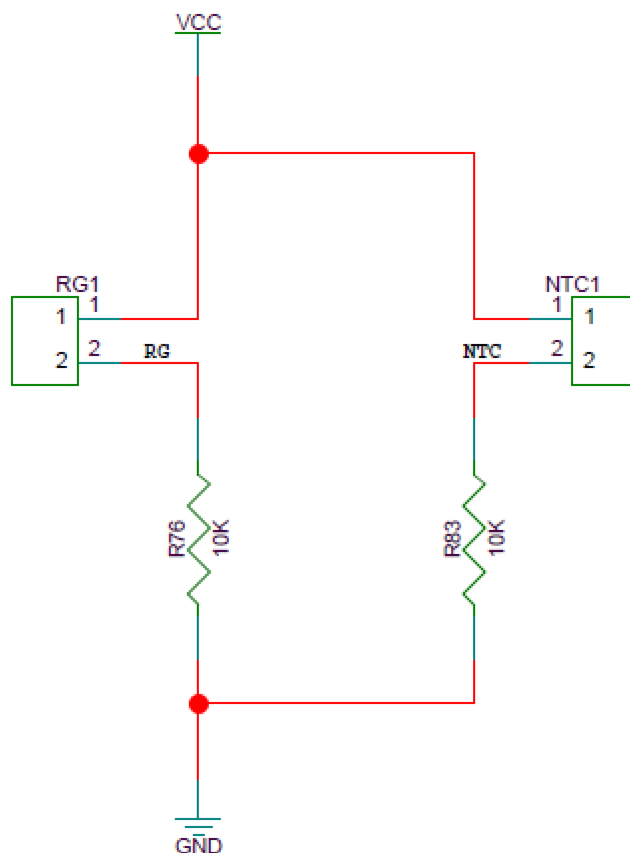


图 3-151 硬件原理图

可以看到原理图，NTC1是热敏电阻，RG1是光敏电阻，电阻一端接5V的VCC，另外一段接一个10K的电阻，电阻的另一端接GND。

这里将RG或者NTC的管脚接到AD/DA芯片PCF8951上，随着热敏和光敏的阻值变化，形成了一个热敏电阻和光敏电阻跟10K电阻之间电压的分配变化，所以形成了一个模拟变量的输入到AD芯片PCF8951里，通过电阻的变化影响电压的变化，来检测热敏或者光敏电阻的变化状态。

3.22.5 例程 01 热敏电阻数码管显示

程序如下：

```
/**********************************************************************
* 例程：PCF8591 热敏电阻值数码管显示
* 作者：www.armjishu.com
* 内容：PCF8591 第 3 路 AD 转换采样热敏电阻的分压值并数码管显示
* 现象：本实验是通过 51 单片机的 IO 管脚模拟 I2C 协议访问 PCF8591 芯片。神舟 51
*       开发板上 PCF8591 第 3 路 AD 连接到热敏电阻 NTC1，转换值显示在数码管上
*       动态显示。将 AD 转换值转换为电压值，范围为 0.0-5.0。改变热敏电阻
*       的温度，看到数码管上的数值随之变化，则说明 PCF8591 访问成功。
**********************************************************************/
/* 包含头文件 */
```

```
#include <reg52.h>
#include "i2c.h"
#include "delay.h"
#include "display.h"
#define AddWr 0x90      //写数据地址
#define AddRd 0x91      //读数据地址
unsigned char ReadADC(unsigned char Chl);
/*------------------------------------------------
                主程序
------------------------------------------------*/
main()
{
        unsigned char num=0;
        Init_Timer0();
        while (1)    //主循环
        {
                //  第 3 路 AD 采样数值
                num=ReadADC(2);
                TempData[1]=DuanMa[num/100];
                TempData[2]=DuanMa[(num%100)/10];
                TempData[3]=DuanMa[(num%100)%10];
                // x5  表示基准电压 5V
                // x10 表示把实际值扩大 10，如 4.5  变成  45  方便做下一步处理
                num = (num*5*10)>>8;
                TempData[6]=DuanMa[num/10]|0x80;      // |0x80 表示显示小数点
                TempData[7]=DuanMa[num%10];
                //主循环中添加其他需要一直工作的程序
                DelayMs(100);
        }
}
/*------------------------------------------------
                    读 AD 转值程序
输入参数  Chl  表示需要转换的通道，范围从 0-3
返回值范围 0-255
------------------------------------------------*/
unsigned char ReadADC(unsigned char Chl)
{
        unsigned char Val;
        Start_I2c();                //启动总线
        SendByte(AddWr);        //发送器件地址  写命令
        if(ack==0)
        {
                return(0);
        }
```

```
    SendByte(0x40|Chl);   //发送控制字节
    if(ack==0)
    {
        return(0);
    }
    Start_I2c();
    SendByte(AddRd);      //发送器件地址 读命令
    if(ack==0)
    {
        return(0);
    }
    Val=RcvByte();          //读 AD 转值程序
    NoAck_I2c();           //发送非应位
    Stop_I2c();            //结束总线
    return(Val);
}
```

由于PCF8591芯片在神舟51开发板中已经将SCL连接到51单片机的P21，SDA连接到51单片机的P20，所以需要连接热敏电阻以及LED数码管。将神舟开发板的JP15排针连接到LED数码管的JP23。

硬件连接关系配置如下表3-104所示：

表3-104 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P0 口	JP15(A 向左)	直连	JP23(B 向左)	8 芯排线	控制 LED 数码管
P22-P24	JP27	跳帽	JP22	8 个跳帽	74LS138 数码管位选
	JP6.AI2	直连	JP30.热敏	1 根杜邦线	热敏电阻分压值
实验现象：本实验是通过 51 单片机的 IO 管脚，模拟 I2C 协议访问 PCF8591 芯片，第三路 AD 采样热敏电阻的分压值，并显示转换结果。神舟 51 开发板的第三路 AD 通过一根杜邦线连接到热敏电阻分压电路，通过改变热敏电阻 NTC1 的温度可以使 PCF8591 得到不同的模拟输入。将 PCF8591 转换值显示在数码管上动态显示。将 AD 转换值转换为电压值，范围为 0.0-5.0。实验改变热敏电阻 NTC1 的温度，看到数码管上的数值随之变化，则说明 PCF8591 访问成功。					

连接图如下图3-152所示：

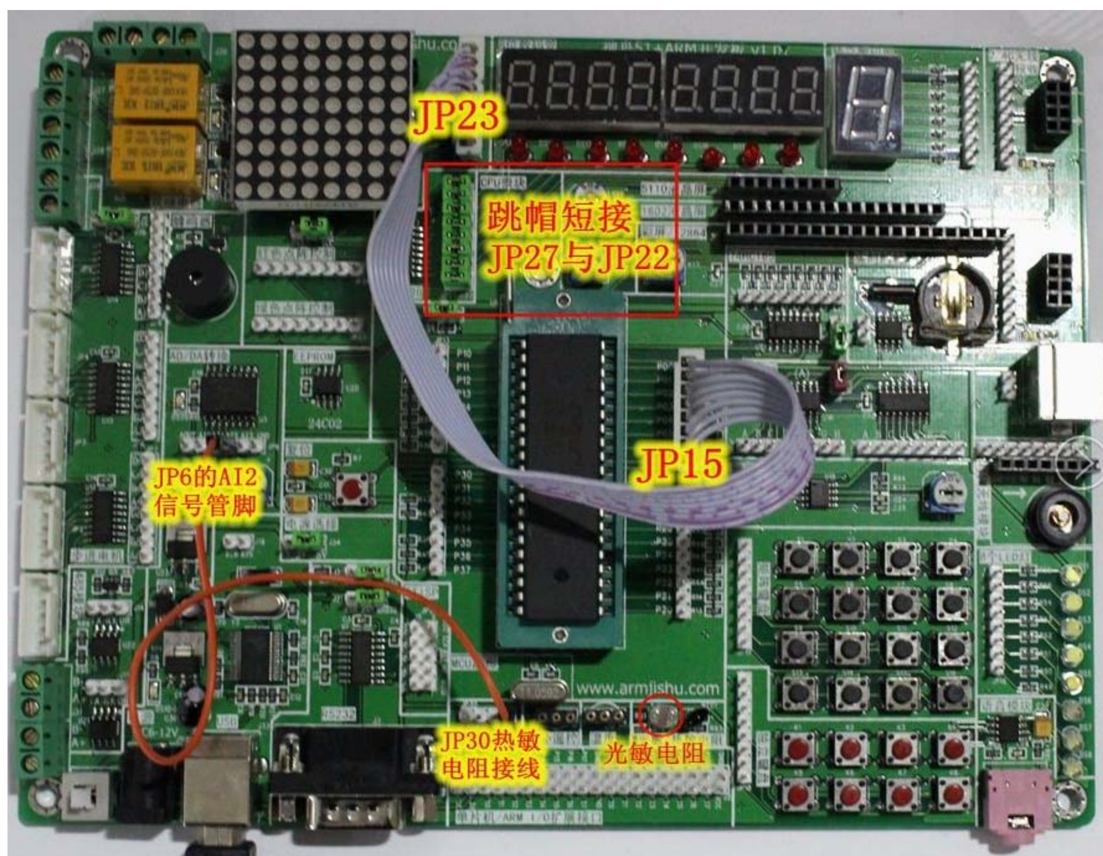


图 3-152 硬件连接实物图

知识要点：

关于 AD/DA 芯片 PCF8951 的使用请参考 AD/DA 的相关章节，里面有详细描述。

3.22.6 例程 02 光敏电阻数码管显示

程序如下：

```
/************************************************************************
* 例程：PCF8591 光敏电阻值数码管显示
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：PCF8591 第 4 路 AD 转换采样光敏电阻的分压值并数码管显示
* 现象：本实验是通过 51 单片机的 IO 管脚模拟 I2C 协议访问 PCF8591 芯片。神舟 51
*       开发板上 PCF8591 第 3 路 AD 连接到光敏电阻 RC1，转换值显示在数码管上
*       动态显示。将 AD 转换值转换为电压值，范围为 0.0-5.0。改变光敏电阻
*       的光照强度，看到数码管上的数值随之变化，则说明 PCF8591 访问成功。
************************************************************************/
/* 包含头文件 */
#include <reg52.h>
#include "i2c.h"
#include "delay.h"
#include "display.h"
```

```
#define AddWr 0x90      //写数据地址
#define AddRd 0x91      //读数据地址
unsigned char ReadADC(unsigned char Chl);

/*------------------------------------------------
                    主程序
------------------------------------------------*/
main()
{
        unsigned char i=0;
        unsigned char num=0;
        unsigned int    value=0;
        Init_Timer0();
        while (1)    //主循环
        {
                //将电压值采样 8 次相加后计算平均值
                value=0;
                for(i=0;i<8;i++)
                {
                    //  第 3 路 AD 采样数值
                    value = value + ReadADC(3);
                    DelayMs(2);
                }
                //计算平均值，右移 3 就等价于除以 8
                num=(value>>3);
                TempData[1]=DuanMa[num/100];
                TempData[2]=DuanMa[(num%100)/10];
                TempData[3]=DuanMa[(num%100)%10];
                // x5  表示基准电压 5V
                // x10 表示把实际值扩大 10，如 4.5  变成  45  方便做下一步处理
                num = (num*5*10)>>8;
                TempData[6]=DuanMa[num/10]|0x80;      // |0x80 表示显示小数点
                TempData[7]=DuanMa[num%10];
                //主循环中添加其他需要一直工作的程序
                DelayMs(200);
        }
}
/*------------------------------------------------
                    读 AD 转值程序
输入参数  Chl  表示需要转换的通道，范围从 0-3
返回值范围 0-255
------------------------------------------------*/
unsigned char ReadADC(unsigned char Chl)
{
```

```
    unsigned char Val;
    Start_I2c();             //启动总线
    SendByte(AddWr);         //发送器件地址  写命令
    if(ack==0)
    {
        return(0);
    }
    SendByte(0x40|Chl);      //发送控制字节
    if(ack==0)
    {
        return(0);
    }
    Start_I2c();
    SendByte(AddRd);         //发送器件地址  读命令
    if(ack==0)
    {
        return(0);
    }
    Val=RcvByte();           //读 AD 转值程序
    NoAck_I2c();             //发送非应位
    Stop_I2c();              //结束总线
    return(Val);
}
```

由于PCF8591芯片在神舟51开发板中已经将SCL连接到51单片机的P21，SDA连接到51单片机的P20，所以需要连接光敏电阻以及LED数码管。将神舟开发板的JP15排针连接到LED数码管的JP23。硬件连接关系配置如下表3-105所示：

表3-105硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P0 口	JP15(A 向左)	直连	JP23(B 向左)	8 芯排线	控制 LED 数码管
P22-P24	JP27	跳帽	JP22	8 个跳帽	74LS138 数码管位选
	JP6.AI3	直连	JP30.光敏	1 根杜邦线	光敏电阻分压值
实验现象：本实验是通过 51 单片机的 IO 管脚，模拟 I2C 协议访问 PCF8591 芯片，第四路 AD 采样光敏电阻的分压值，并显示转换结果。神舟 51 开发板的第四路 AD 通过一根杜邦线连接到光敏电阻分压电路，通过改变光敏电阻 RC1 的光照强度可以使 PCF8591 得到不同的模拟输入。将 PCF8591 转换值显示在数码管上动态显示。将 AD 转换值转换为电压值，范围为 0.0-5.0。实验改变光敏电阻 RC1 的温度，看到数码管上的数值随之变化，则说明 PCF8591 访问成功。					

连接图如下图3-153所示：

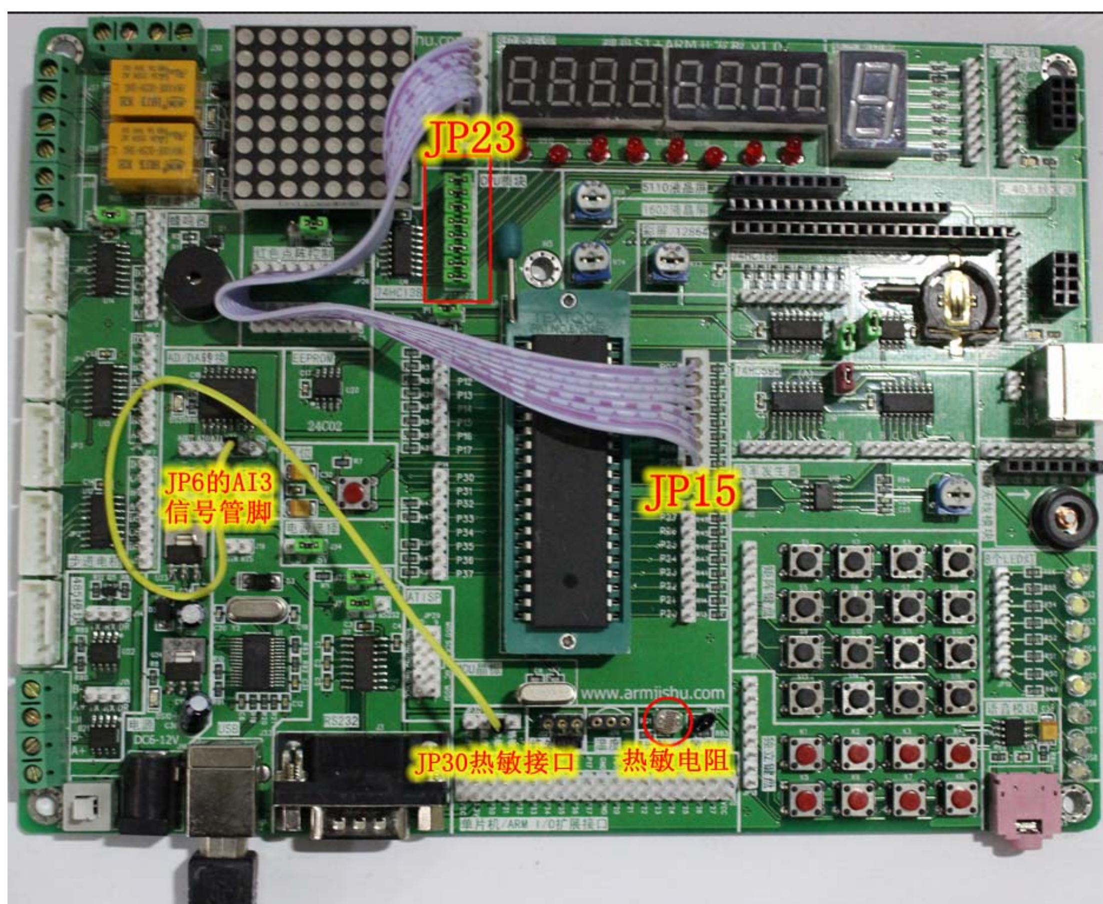


图 3-153 硬件连接实物图

知识要点：

关于AD/DA芯片PCF8951的使用请参考AD/DA的相关章节，里面有详细描述。

3.23 RS-485 通讯

3.231 串行通讯

随着计算机网络化和微机分级分布式应用系统的发展，通信的功能越来越重要。通信是指计算机与外界的信息传输。通信方式可分为：并行通信和串行通信。这里我们的 RS485 使用的是串行通讯，所以我们下面讲下串行通讯。

串行通讯是指使用一条数据线，将数据一位一位地依次传输，每一位数据占据一个固定的时间长度。其只需要少数几条线就可以在系统间交换信息，特别使用于计算机与计算机、计算机与外设之间的远距离通信，一条信息的各位数据被逐位按顺序传送的通讯方式称为串行通讯。

串行通讯的特点是：数据位传送，即数据传送按位顺序进行，最少只需一根传输线即可完成，成本低但送速度慢。串行通讯的距离可以从几米到几千米。根据信息的传送方向，串行通讯可以进一步分为单工、半双工和全双工三种。信息只能单向传送为单工；信息能双向传送但不能同时双向传送称为半双工；信息能够同时双向传送则称为全双工。 串行通讯又分为异步通讯和同步通讯两种方式。在单片机中，主要使用异步通讯方式。

3.23.2 RS-485 串行通讯介绍

RS-485 是一种通信方式，一种通信协议。最初收发节点是数据模拟信号输入输出简单的信号，后来仪表接口很多都采用了 RS232 接口，也就是现在的串口通信，这种接口虽然可以实现点对点的通信方式，但不足之处是这种方式不能实现联网功能。随后出现的 RS485 解决了这个问题，不仅能实现节点与节点之间的单点通信，还能实现多个节点之间的联网。

RS485 的通信时逻辑“1”以两线间的电压差为+(2-6)V 表示；逻辑“0”以两线间的电压差为-(2-6)V 表示，所以 RS-485 收发器采用平衡发送和差分接收(电压正负相抵最大有 12V 的范围，而普通 TTL 电平一般只有 0-5V 的差额)，因为差额比较大，所以抵抗干扰的能力比较强，可靠通信的传输距离可达数千米。使用 RS-485 总线组网，只需一对双绞线就可实现多系统联网构成分布式系统、设备简单、价格低廉、通信距离长。

3.23.3 MAX485 收发器芯片介绍

MAX485 接口芯片是 Maxim 公司的一种 RS－485 芯片。采用单一电源+5 V 工作，额定电流为 300 μA，采用半双工通讯方式。它完成将 TTL 电平 与 RS－485 电平转换的功能。我们的神舟 51+ARM 单片机用到的就是这款芯片.，它具有一个驱动器和一个接收器。驱动器摆率不受限制，可以实现最高 2.5Mbps 的传输速率。MAX485 通讯程序与 MAX232 通讯程序在本质上是一样的，只是 MAX485 通讯程序需要加上通讯方向控制

其引脚结构图如下图 3-156 所示。从图中可以看出,MAX485 芯片的结构和引脚都非常简单,内部含有一个驱动器和接收器。

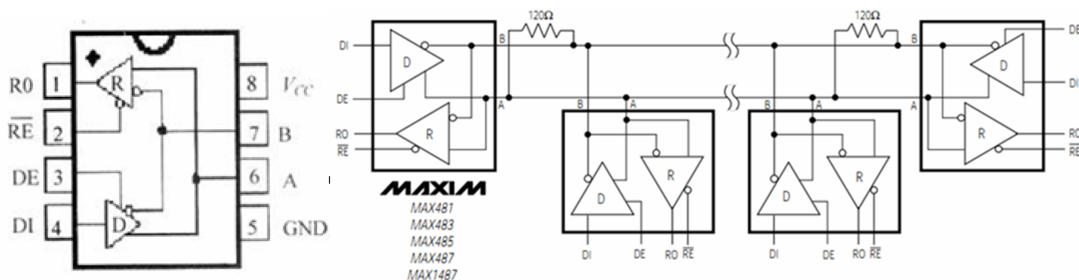


图 3-156 MAX485 引脚和结构图

RO——输出端 （与单片机的 RXD 相连）

/RE——输出使能端（/RE 为逻辑 0 时，器件处于接收状态）

DI——接收端 （与单片机的 TXD 相连）

DE——接收使能端（当 DE 为逻辑 1 时，器件处于发送状态）

A——接收的差分信号端,

B——发送的差分信号端

VCC——电源端

GND——地端

注:（1）/RE、DE（通常接一个引脚即可 0—接收 、1—发送 ），因为 MAX485 工作在半双工状态，所以只需用单片机的一个管脚控制这两个引脚即

（2）当 A 引脚的电平高于 B 时，代表发送的数据为 1；当 A 的电平低于 B 端时，代表发送的数据为 0。在与单片机连接时接线非常简单。只需要一个信号控制 MAX485 的接收和发送即可。同时将 A 和 B 端之间加匹配电阻，一般可选 100Ω的电阻

（3）如需要了解更详细的 MAX485 芯片资料，请浏览芯片用户手册资料。

RS-485 与 RS-232 不一样，数据信号采用差分传输方式，也称作平衡传输，它使用一对双绞线，将其中一线定义为 A，另一线定义为 B。通常情况下，发送驱动器 A、B 之间的正电平在+2~+6V，是一个逻 辑状态，负电平在-2V~6V，是另一个逻辑状态。另有一个信号地 C，在 RS-485 中还有一“使能”端，而 在 RS-422 中这是可用可不用的。“使能”端是用于控制发送驱动器与传输线的切断与连接。当“使能” 端起作用时，发送驱动器处于高阻状态，称作“第三态”，即它是有别于逻辑“1”与“0”的第三态。

3.23.4 RS-485 网络通讯结构

RS-485 采用半双工工作方式，任何时候只能有一点处于发送状态，因此，发送电路须由使能信号加以控制，RS-485 在用于多点互连时非常方便，可以省掉许多信号线。应用 RS-485 可以联网构成分布式系统。

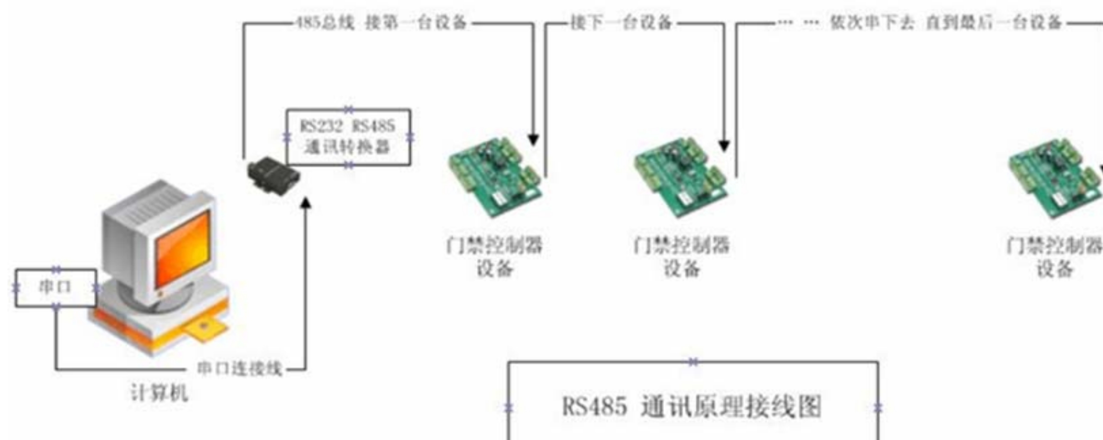


图 3-154 RS485 级联电路接线图

485 总线要采用手拉手结构，而不能采用星形结构。如下图 3-154，如果有星型连接或者分叉，干扰将非常大，通讯不畅，甚至通讯不上。下图 3-155 是常见的错误的连接方式：

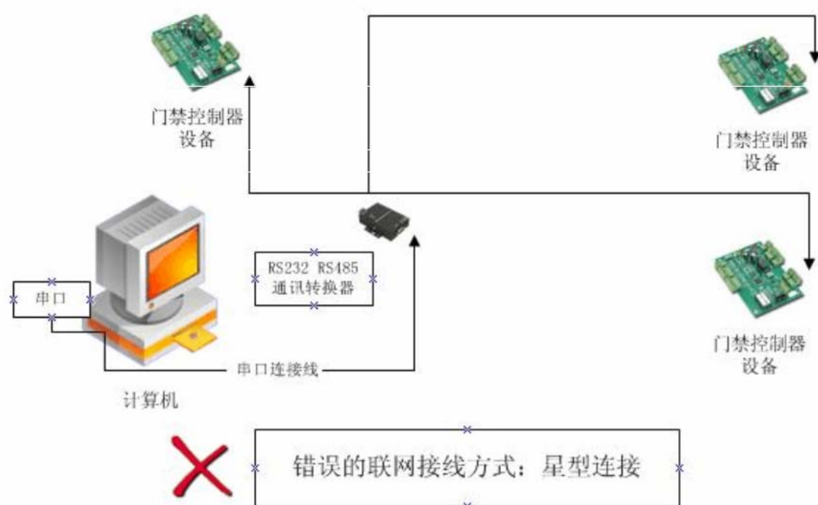


图 3-155 错误连接方式

3.23.6 硬件原理图说明

硬件连接的原理图如下图 3-157 所示：

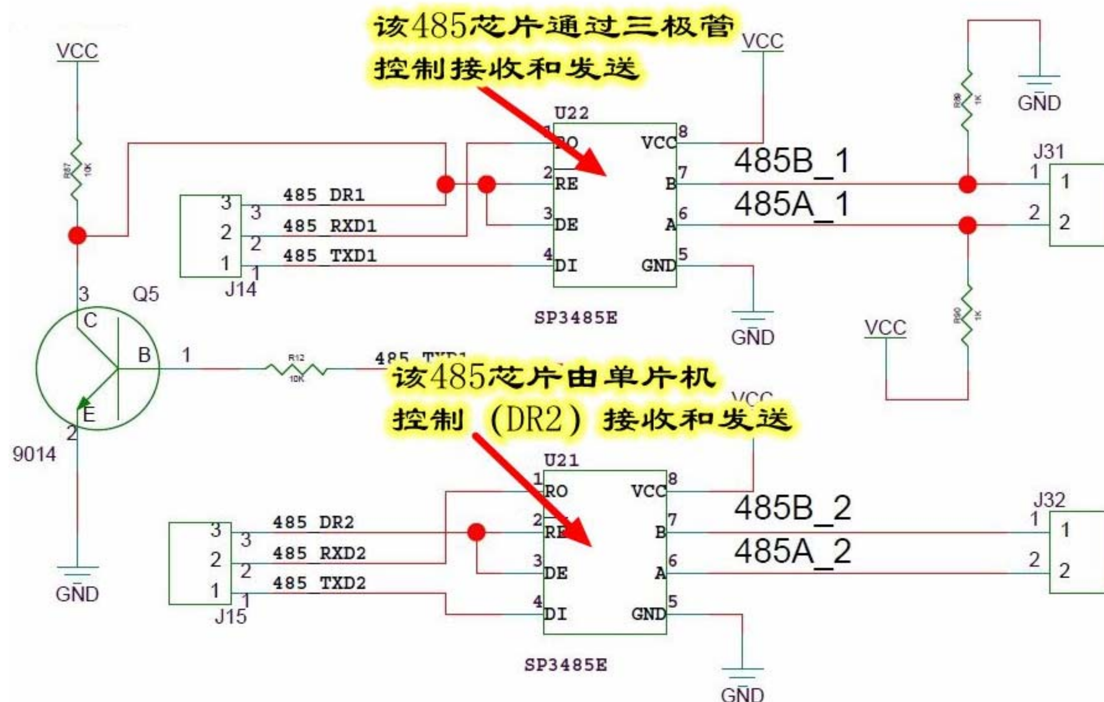


图 3-157 硬件连接原理图

图中，三极管 Q5 控制 U22 芯片的工作状态，如 1 为发送，0 为接收模式，而三极管受 485_TXD1 控制，它为高电平时，485 芯片处于接收模式，为低电平时，485 芯片处于发送模式。U21 的 485 芯片工作方式由单片机(DR2)控制。

3.23.7 例程 01 RS485 通讯实验

代码如下：

```
/*******************************************************
* 例程：串口通信 RS485
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：485 与 232 使用相同软件协议，是半双工，
        需要有 2 套开发板对发测试，或者使用另外一
        个 485 设备配套，这里提供测试程序
*******************************************************/
#include<reg52.h> //包含头文件，一般情况不需要改动，头文件包含特殊功能寄存器的
定义
/*------------------------------------------------
          硬件端口定义
------------------------------------------------*/
sbit Ctrl_EN = P3^7;     //发送接收控制端
/*------------------------------------------------
            函数声明
------------------------------------------------*/
void SendStr(unsigned char *s);

/*------------------------------------------------
 uS 延时函数，含有输入参数 unsigned char t，无返回值
 unsigned char 是定义无符号字符变量，其值的范围是
 0~255 这里使用晶振 12M，精确延时请使用汇编,大致延时
 长度如下 T=tx2+5 uS
------------------------------------------------*/
void DelayUs2x(unsigned char t)
{
    while(--t);
}
/*------------------------------------------------
 mS 延时函数，含有输入参数 unsigned char t，无返回值
 unsigned char 是定义无符号字符变量，其值的范围是
 0~255 这里使用晶振 12M，精确延时请使用汇编
------------------------------------------------*/
void DelayMs(unsigned char t)
{
    while(t--)
    {
        //大致延时 1mS
        DelayUs2x(245);
        DelayUs2x(245);
```

```
        }
}
/*------------------------------------------------
                    串口初始化
------------------------------------------------*/
void InitUART    (void)
{
        SCON    = 0x50;                         // SCON:  模式  1, 8-bit UART,  使能接收
        TMOD |= 0x20;                           // TMOD: timer 1, mode 2, 8-bit  重装
        TH1      = 0xFD;                         // TH1:    重装值  9600  波特率  晶振  11.0592MHz
        TR1      = 1;                           // TR1:    timer 1  打开
        EA       = 1;                           //打开总中断
}
/*------------------------------------------------
                          主函数
------------------------------------------------*/
void main (void)
{
        InitUART();
        Ctrl_EN=1;      //发送模式    Ctrl_EN=1 为发送，0 为接收模式
        while (1)
                {
                        SendStr("\r\n  神舟 51 单片机开发板  串口测试!");
                        DelayMs(240);//延时循环发送
                        DelayMs(240);
                }
}
/*------------------------------------------------
                        发送一个字节
------------------------------------------------*/
void SendByte(unsigned char dat)
{
        SBUF = dat;
        while(!TI);
        {
                TI = 0;
        }
}
/*------------------------------------------------
                      发送一个字符串
------------------------------------------------*/
void SendStr(unsigned char *s)
{
        while(*s!='\0')       // \0  表示字符串结束标志，通过检测是否字符串末尾
```

```
        {
            SendByte(*s);
            s++;
        }
    }
```

硬件连接实物图如下图 3-158 所示：

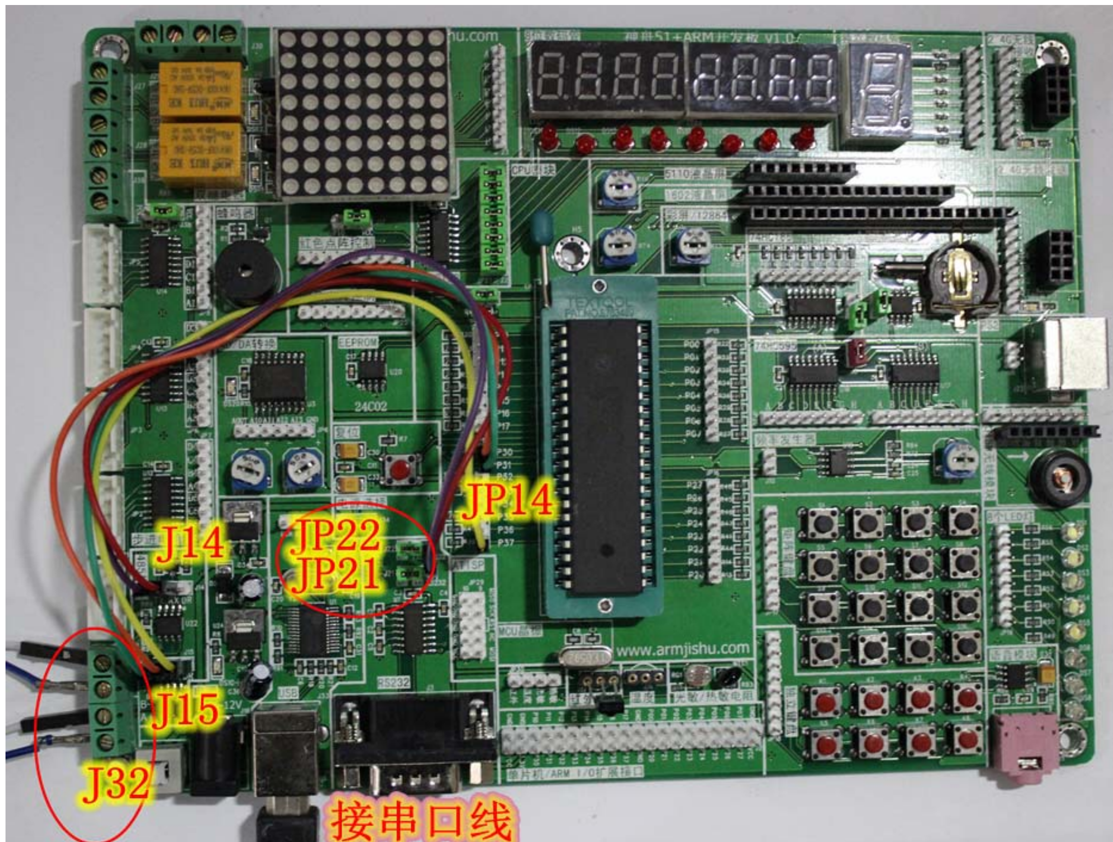


图 3-158 硬件连接实物图

连接对应关系表 3-106 如下：

表 3-106 硬件连接关系

用排线电缆或杜邦线连接“单片机 IO”和“模块接口”					
单片机接口	插座 1	方式	插座 2	线缆	功能
P3	JP14 的 P30	直连	J15 的 RX	1 根单针杜邦线	单片机串口的 TX
P3	JP14 的 P31	直连	J15 的 TX	1 根单针杜邦线	单片机串口的 RX
P3	JP14 的 P37	直连	J15 的 DR	1 根单针杜邦线	单片机控制 485 模块 1 的接收发送
	J22 的 RS232	直连	J14 的 RX	1 根单针杜邦线	485 模块 2 的 RX 与 DB9 串口的 TX 连接
	J21 的 RS232	直连	J14 的 TX	1 根单针杜邦线	485 模块 2 的 TX 与 DB9 串口的 RX 连接
	J31 的 B-	直连	J32 的 B-	1 根单针杜邦线	2 个 485 直连
	J31 的 A+	直连	J32 的 A+	1 根单针杜邦线	2 个 485 直连
J22 短接 USB 端，J21 短接 USB 端					
实验现象： 下载程序后，我们可以通过 STC_ISP_V488.exe 软件的串口工具，设置好波特率和串口					

号，打开串口就能打印出“神舟 51 单片机开发板 串口测试!”。如下图 3-162

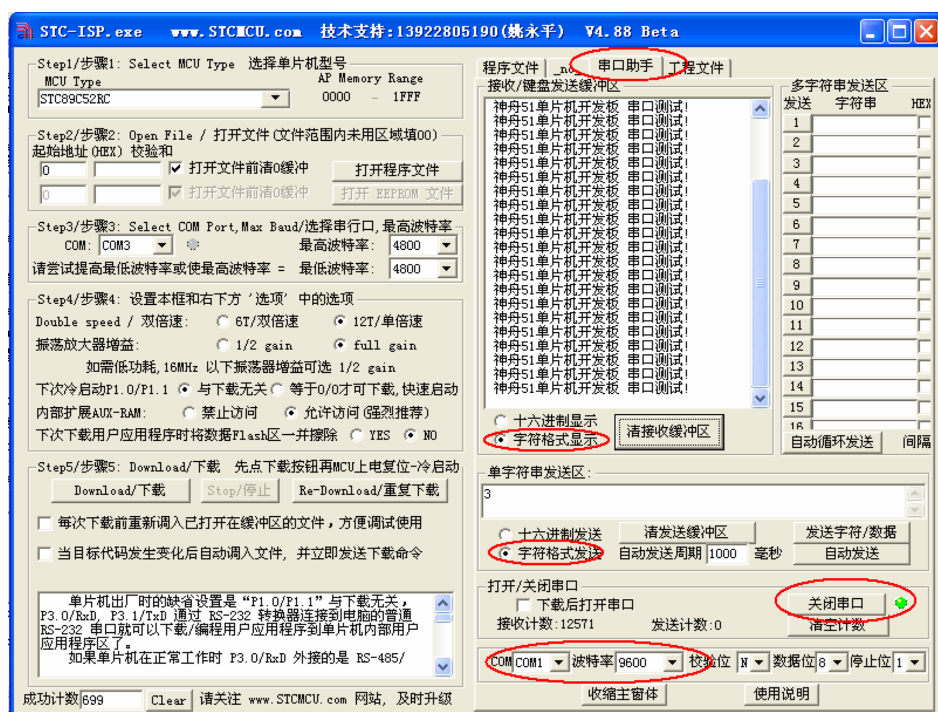


图 3-159 串口助手

知识要点：

1、通过单片机控制 RS485 芯片的 DR 脚来控制数据的接收和发送即“Ctrl_EN”；更多实验可通过串口例程进行修改；只需增加控制 485 的使能 Ctrl_EN=1 或 Ctrl_EN=0（1 为发送，0 为接收模式）。

2、本例子中需要使用到的 DB9 串口线为交叉双母头。

3.24 18B20 温度传感器

3.24.1 简介

温度是表示物体冷热程度的物理量，最常见的物理量之一，如：气温、体温、水温、油温、锅炉温度、电器温度等。随着科学技术的发展，对温度的测量也是多种多样。

温度传感器可以传导温度，它可以代替人体去感受外界的温度，并且能够把一个温度数字化，具体化，避免了人体去直接接触感觉温度，过低或者过高的温度可能会冻伤或者是烧伤人体，方便人类去管理和控制调整温度的变化。日常生活中各个领域中都使用到温度传感器，像家庭中的电子温度计、工厂检测机器温度的仪器、交通工具上的等等。它为人类带来了很大的便利。

3.24.2 各种温度测量方法

利用各种物质本身的物理特性，根据不同温度条件下该物理特性的变化，制造出不同的

温度测试工具。下面举一些各种温度测量方法：

酒精温度计。利用酒精热胀冷缩的性质制成的温度计，也是最常见的环境温度计，外壳透明，内部红色酒精温度条；其成本和安全性比水银温度计高，一般测量温度范围是 -114℃ ～78℃，可满足测量体温和气温的要求。

水银温度计。与酒精温度计类似，利用水银的热胀冷缩制成；水银的冰点是:-39℃,沸点是:356.7℃，其冰点相对酒精要低，所以对于低温环境，北极、珠穆朗玛峰等不适用；但其较高的沸点，高精度，通常用来做科学实验和测量人体温度等。

热电阻温度计。热电阻是中低温区最常用的一种温度检测器。它的主要特点是测量精度高，性能稳定。其中铂热是阻的测量精确度是最高的，它不仅广泛应用于工业测温，而且被制成标准的基准仪，医疗方面也可作为电子体温计。一般测量温度范围为 -200℃~800℃。

热电偶温度计。热电偶是温度测量仪表中常用的测温元件，是由两种不同成分的导体两端接合成回路时，当两接合点热电偶温度不同时，就会在回路内产生热电流。其测温范围一般为 -200℃～1300℃，特殊情况下可高达 －270℃～2800℃。相对于热电阻，热电偶测量精度一般不如热电阻，但是其测温范围更宽（特别是高温部分），测量速度快，能够远传 4-20mA 电信号，便于自动控制和集中控制

红外温度计。某些安检进出口有用到这种温度计，红外测温仪采用非接触红外传感技术对目标进行安全、准确、快速、可靠的测量。 红外测温的原理：自然界中一切温度高于绝对零度（-273.15℃）的物体都会辐射出红外线，而辐射出的红外线的能量和温度是成正比的关系，红外测温仪就是通过透镜（如菲涅尔透镜）收集并汇集红外能量到红外传感器上，将其转化成一个电压信号，标定此电压与实际温度的对应关系，即可得到所测目标温度值。目前红外测温仪及应用系统，已广泛应用于测量机械、化工、陶瓷、轻工、食品、冶金、电力、热处理等行业高温、危险及难以接近物体表面的温度。

3.24.3 什么是温度传感器

温度传感器是指能感受温度并把温度这样的模拟值转换成数字信号的传感设备，通过数字信号来描述温度的高与低。

我们都知道温度理论上是无上限，也无下限的；例如火山或者太阳的温度可以非常高，南北极的冰雪温度也可以非常低；所以温度范围是非常广乏的，而目前市场上比较关心的温度范围主要是 xx~xx 摄氏度；根据温度变化的灵敏度和精度，又有很多不同温度传感器的种类。例如，家庭用的一般是 0 到 100 摄氏度的温度计，钢铁厂用的一般是几千度摄氏度的温度计。

另外温度传感器还有一个参数叫灵敏度和精度，那什么是灵敏度呢？灵敏度例如人体感知温度变化需要 1 秒钟，而温度传感器感知温度变化只需零点几秒就可以，这个时间就是灵敏度；那什么是精度呢？有的传感器最小描述单位是 0.5 摄氏度，有的传感器最小描述单位是 0.1 摄氏度。

温度传感器是如何感知温度的？这就要看温度传感器是什么材料制作以及制作的原理。比如有的材料可以感知常温下的温度，但是如果温度超过 300 摄氏度，这个材料有可能会被融化；有的材料就可以耐高温，无论任何种类的具体材料都会有一个属于自己的熔点，超过这个熔点温度就会融化，所以每个传感器的可测温度的值是受材料或者其他因素影响和限制的，也就是说，温度可测区间是有范围的；但是这个世界的传感器技术每天都在进步和发展，对于外界温度的感知技术也是在不断进步的，或许有一天，我们能有一个传感器能进入到太阳的内部去检测它的温度也说不定。

温度传感器在不同的温度环境下，内部的温度感知材料会产生不同的物理特性，通过观

察这些物理特性的特点，从而得出当前的一个温度判断；再把这个温度判断通过数字电平的方式输出，或者通过某种模拟值来表达，这就使得所有的温度传感器都会有一个说明书，说明书上会写这些温度传感器的特性，某种特性对应某个温度的一个公式或者列表。

3.24.4 18B20 温度传感器的实现原理

18B20 温度传感器能把读到的温度转成二进制的格式发送给单片机，单片机收到它发过来的数据后再转换成比如十进制的方式让我们能知道当前的温度是多少。它的测温范围是-55～125 ℃。固有测温分辨率为 0.5 ℃。在这个范围内的温度它都能测试出来。

我们先来看下下图 3-160 18B20 温度传感器内部的测温原理图，通过该图从传感器内部分析实现测温原理：

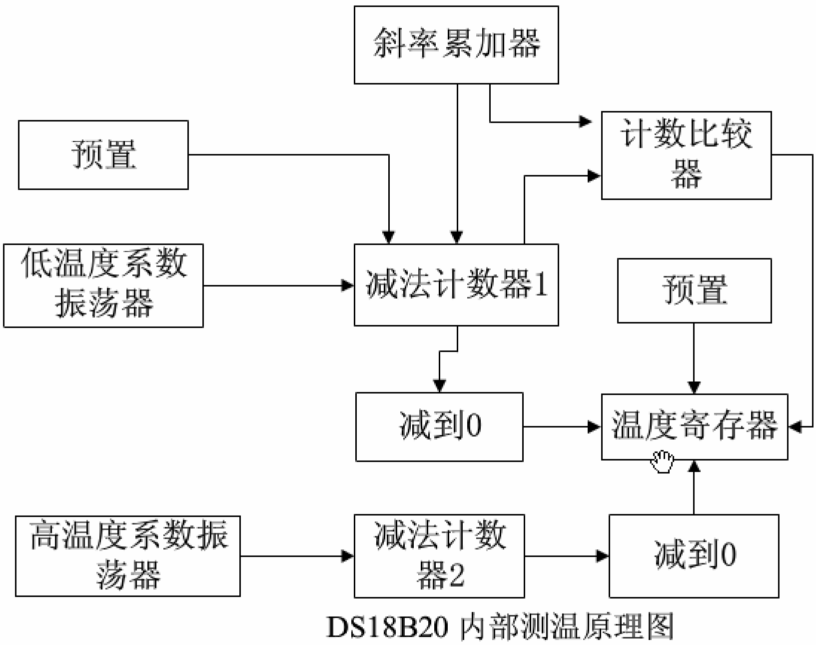


图 3-160 18B20 温度传感器内部的测温原理图

上图中的低温度系数晶振的振荡频率受温度的影响很小，被用于产生固定频率的脉冲信号送给减法计数器 1；高温度系数晶振的振荡频率随温度变化而明显改变，所产生的信号作为减法计数器 2 的脉冲输入，所以温度的检测主要是靠高温度系数晶振来反馈；图中还隐含着计数门，当计数门打开时，DS18B20 就对低温系数振荡器产生的时钟脉冲进行计数，进而完成温度测量。计数门的开启时间由高温度系数振荡器决定，每次测量前，首先将-55°C 所对应的基数分别置入减法计数器 1 和温度寄存器中，减法计数器 1 和温度寄存器被预置在-55°C 所对应的一个基数值。减法计数器 1 对低温度系数晶振产生的脉冲信号进行减法计数，当减法计数器 1 的预置值减到 0 时温度寄存器的值将加 1，减法计数器 1 的预置将重新被装入，减法计数器 1 重新开始对低温度系数晶振产生的脉冲信号进行计数，如此循环直到减法计数器 2 计数到 0 时，停止温度寄存器值的累加，此时温度寄存器中的数值即为所测温度。即当减法计算器 2 中为 0 时，减法计算器 1 被循环了多少次，这个次数以-55 度往上加，就是被检测的温度了，通过这个规律来判断温度的值。

3.24.5 18B20 温度传感器ROM存储器

传感器的指令是与外界沟通的桥梁，可以通过这些指令来达到控制传感器的目的。DS18B20 温度传感器的内部存储器包括一个高速暂存 RAM 和一个非易失性的可电擦除的 ROM（64 位），本节先分析 ROM。

ROM 的 64 位结构是：开始 8 位（地址：28H）是产品类型标号，接着的 48 位是该 DS18B20 自身的序列号，并且每个 DS18B20 的序列号都不相同，因此它可以看作是该 DS18B20 的地址序列码；最后 8 位则是前面 56 位的循环冗余校验码。由于每一个 DS18B20 的 ROM 数据都各不相同，因此微控制器就可以通过单总线对多个 DS18B20 进行寻址，从而实现一根总线上挂接多个 DS18B20 的目的；DS18B20 温度传感器 64 位闪速 ROM 的结构如表 3-107：

表 3-107 闪速 ROM 的结构

8 位 检验 CRC	48 位 序列号	8 位 工厂代码（10H）

ROM 存放高温度和低温度触发器 TH、TL 和结构寄存器。数据先写入 RAM，经校验后再传给 ROM。可以通过 ROM 指令和 RAM 指令来实现对 18B20 传感器的控制。

配置寄存器为高速暂存器中的第 5 个字节，他的内容用于确定温度值的数字转换分辨率，DS18B20 工作时按此寄存器中的分辨率将温度转换为相应精度的数值。该字节各位的定义如表 3-108：

表 3-108　DS18B20 内部存储器

TM	R1	R0	1	1	1	1	1

低 5 位一直都是 1，TM 是测试模式位，用于设置 DS18B20 在工作模式还是在测试模式。在 DS18B20 出厂时该位被设置为 0，用户不要去改动，R1 和 R0 决定温度转换的精度位数，即是来设置分辨率，如表 3-109 所示（DS18B20 出厂时被设置为 12 位）。

表 3-109 R1 和 R0 模式表

R1	R0	分辨率	温度最大转换时间/mm
0	0	9 位	93.75
0	1	10 位	187.5
1	0	11 位	275.00
1	1	12 位	750.00

由表 3-101 可见，设定的分辨率越高，所需要的温度数据转换时间就越长。因此，在实际应用中要在分辨率和转换时间权衡考虑。

DS18B20 完成温度转换后，就把测得的温度值与 TH，TL 作比较，若 T>TH 或 T＜TL,则将该器件内的告警标志置位，并对主机发出的告警搜索命令作出响应。因此，可用多只 DS18B20 同时测量温度并进行告警搜索。

CRC 的产生在 64 位 ROM 的最高有效字节中存储有循环冗余校验码（CRC）。主机根据 ROM 的前 56 位来计算 CRC 值，并和存入 DS18B20 中的 CRC 值做比较，以判断主机收到的 ROM 数据是否正确。

3.24.6 18B20 温度传感器RAM存储器

高速暂存存储器 RAM 由 9 个字节组成，当温度转换命令发布后，经转换所得的温度值以二字节补码形式存放在高速暂存存储器的第 0 和第 1 个字节。单片机可通过单线接口读到该数据，读取时低位在前，高位在后，对应的温度计算：当符号位 S=0 时，直接将二进制位转换为十进制;当 S=1 时，先将补码变为原码，再计算十进制值，RAM 其分配如下表 3-110 所示。

表 3-110　RAM 结构分配表

温度低位	温度高位	TH	TL	配置	保留	保留	保留	8 位 CRC

其中温度信息（第 1，2 字节）、TH 和 TL 值第 3，4 字节、第 6～8 字节未用，表现为全逻辑 1；第 9 字节读出的是前面所有 8 个字节的 CRC 码，可用来保证通信正确。

当传感器完成温度测量后（它的测量精度可以配置成 9 位，10 位，11 位或 12 位四种状态），先将得到的结果存储在 RAM 的前两个字节中，即两个 8BIT 的 RAM 中，这两个 8BIT 存储格式如下表（以 12 位转化为例）：

表 3-111　温度值格式

2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
S	S	S	S	S	2^6	2^5	2^4

这是 12 位转化后得到的 12 位数据，在 RAM 中的的两个 8 比特的存储空间中，二进制中的前面 5 位是符号位，如果测得的温度大于 0，这 5 位为 0 ，则只要将测到的数值乘于 0.0625 即可得到实际温度；如果温度小于 0，这 5 位为 1，测到的数值需要取反加 1 再乘于 0.0625 即可得到实际温度。

例如：+125℃的数字输出为 07D0H，+25.0625℃;的数字输出为 0191H，-25.0625℃的数字输出为 FF6FH，-55℃；的数字输出为 FC90H；温度值格式如表 3-111：

表 3-112　部分温度值

温度/℃	二进制表示		十六进制表示
+125	00000111	11010000	07D0H
+25.0625	00000001	10010001	0191H
+0.5	00000000	00001000	0008H
0	00000000	00000000	0000H
－0.5	11111111	11111000	FFF8H
－25.0625	11111110	01101111	FE6FH
－55	11111100	10010000	FC90H

单片机可通过单线接口读到该数据，读取时低位在前，高位在后。

3.24.7 18B20 温度传感器的工作流程

DS18B20 能直接读出被测温度，并且可根据实际要求通过简单的编程分别在 93.75ms 和 750ms 内实现 9~12 位的数字值读数方式，并且读出或写入仅需要一根信号线（单线接口），因为是一根信号线，所以单线通信功能是分时完成的，它有严格的时隙概念，对 DS18B20 的各种操作必须按协议进行；根据 DS18B20 的协议规定，微控制器控制 DS18B20 完成温度的转换必须经过以下 3 个步骤：1.对 18B20 复位初始化；2.发送 ROM 指令；3.最后发送 RAM 指令，这是一个完整的流程；每次读取都要经过这么 3 个步骤

例如，现在需要读取一个温度值，那么具体的流程如下，后面的例程会有实际代码分析：

第 1 步：复位初始化。

每次读写前都要对 DS18B20 进行复位初始化。复位要求主 CPU 将数据线下拉 500us，然后释放，DS18B20 收到信号后等待 16us~60us 左右，然后发出 60us~240us 的存在低脉冲，主 CPU 收到此信号后表示复位成功。

第 2 步：发送 ROM 指令

忽略 64 位 ROM 地址，直接向 DS18B20 发温度变换命令 (CCH)，后面释放总线至少一秒，让 DS18B20 完成转换的操作，在这里要注意的是每个命令字节在写的时候都是低字节先写，例如 CCH 的二进制为 11001100，在写到总线上时要从低位开始写，写的顺序是“零、零、壹、壹、零、零、壹、壹”。

第 3 步：发送 RAM 指令

主机发出读取 RAM 的命令(BEH)，这个命令是读取 RAM 里 9 个字节的前 2 个字节，前 2 个字节就是温度值，即读完第 0 和第 1 个数据后就不再理会后面 DS18B20 发出的数据即可，同样读取数据也是低位在前的。

3.24.8 18B20 硬件原理图分析

在硬件上，DS18B20 与单片机的连接有两种方法，一种是 Vcc 接外部电源，GND 接地，I/O 与单片机的 I/O 线相连；另一种是用寄生电源供电，此时 VDD、GND 接地，I/O 接单片机 I/O。无论是内部寄生电源还是外部供电，I/O 口线接一个上拉电阻即可。温度传感器硬件原理图如图 3-161 所示：

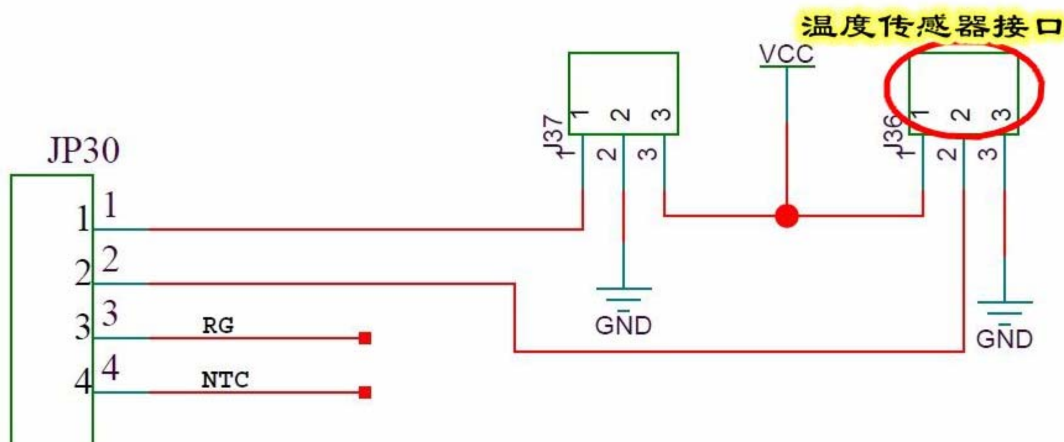


图 3-161 硬件原理图

使用神舟 51+ARM 单片机进行测试，通过单片机板子的 JP30 单排插针与温度传感器进行连接，单片机收到温度传感器发送过来的信号后，把该信号发送到各种显示设备通过十进制的模式显示出来。从图中我们可以看到，温度传感器只需要一根引脚即可完成单片机与它之间的通信。

3.24.9 例程 01 18B20 初始化程序

代码分析：

```
void main(void)
{
    /* 初始化 18B20 传感器 */
    bit dat = 0;
    DQ = 1;  //DQ 复位
    DelayUs2x(5);  //稍微延时
    DQ = 0;          //单片机将 DQ 拉低
    DelayUs2x(200); //精确延时大于 480us 小于 960us
    DelayUs2x(200);
    DQ = 1;          //拉高总线
    DelayUs2x(50);  //15--60us 后接收 60--240us 的存在脉冲
    dat = DQ;        //如果 dat =1 则初始化成功，反之则初始化失败
    DelayUs2x(25);   //稍作延时返回
    /* 初始化 18B20 传感器 */
    while(1)
    {
        if(dat == 0)   //如果 18B20 初始化不正常，LED 灯就不亮
        {
            LED = 1;
        }
        else  //如果 18B20 初始化成功，LED 灯不停的闪烁
        {
            LED = 1;
            DelayUs2x(60000);
            LED = 0;
            DelayUs2x(60000);
        }
    }
}
```

硬件连接关系配置如下表 3-114 所示：

表 3-114 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P1.0	JP13	直连	JP30	1 位杜邦线	‘温度’排针
P1.1	JP13	直连	JP19	1 位杜邦线	‘A’
实验现象：如果 18B20 没有插上，则 LED 灯不亮；如果 18B20 初始化成功，LED 就会不停的闪烁；18B20 的方向是圆弧方向朝 51 单片机座。					

知识要点：

我们看下例程初始化的部分，如下图 3-162：

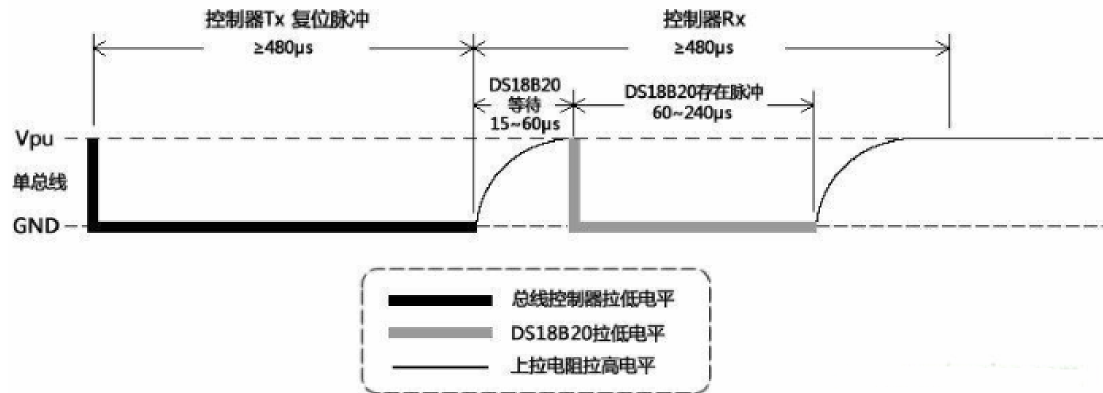


图 3-162 初始化时序

主要初始化过程如下：

1） 首先拉低信号线延时 480--960us，使它复位；
2） 如果成功 18B20 会在 16--60us 时间内有一个从低电平到高电平的变化，我们程序中主要是来读到这个高电平状态，如果读取成功就表示初始化成功，如果读取到是 0 就是表示初始化失败；
3） 初始化成功之后 18B20 内部会自动拉低总线 60—240us，然后它释放总线。

代码中再判断成功是否等于 1，等于 1 表示初始化成功，反之则初始化失败，我们可以看到等待 60—240us 过后，应该是拉高的了。

程序中通过判断这个标志，来控制 LED 灯的亮灭来表示是否初始化成功。

3.24.10 例程 02 18B20 温度采集在 1602 液晶屏上显示

代码分析：

```
void main (void)
{
    int temp;
    float temperature;
    char displaytemp[16];       //定义显示区域临时存储数组
    LCD_Init();                 //初始化液晶
    DelayMs(20);                //延时有助于稳定¨
    LCD_Clear();                //清屏
    Init_Timer0();
    UART_Init();
    Lcd_User_Chr();             //写入自定义字符
    LCD_Write_String(0,0," I love my job ");
    LCD_Write_Char(13,1,0x01);      //写入温度右上角点
    LCD_Write_Char(14,1,'C');   //写入字符 C
    while (1)               //主循环
    {
        if(ReadTempFlag==1)
        {
            ReadTempFlag=0;
```

```
            temp=ReadTemperature();
            temperature=(float)temp*0.0625;
            sprintf(displaytemp,"Temp    % 7.3f",temperature);    //打印温度值
            LCD_Write_String(0,1,displaytemp);      //显示第二行
        }
    }
}
```

知识要点：

该例程实现在 1602 液晶屏上显示温度参数，通过 while()主循环，使用定时器中断，每隔一段时间就会去读取一次 18B20 上的温度值，然后显示在 1602 液晶屏上。

1.进入 main()函数之后，首先通过 LCD_Init()函数对液晶屏初始化，LCD_Clear()对液晶进行清屏操作；Init_Timer0()初始化定时器 0，UART_Init()初始化串口接口，然后对液晶屏写入一些自定义字符，就开始进入 while(1)死循环中，循环定时读取 18B20 的温度值显示在 1602 液晶屏上。

2.接下来详细分析 while()函数，在 while()函数里通过 ReadTempFlag 标志来控制是不是从 18B20 处读一次温度请求并显示在 1602 液晶屏上；ReadTempFlag 标志控制每隔一段时间再读一次温度，ReadTempFlag 值在中断中改变，查看如下定时器 0 的代码

```
void Timer0_isr(void) interrupt 1
{
    static unsigned int num;
    TH0=(65536-2000)/256;          //重新复制 2ms
    TL0=(65536-2000)%256;
    num++;
    if(num==300)             //
    {
        num=0;
        ReadTempFlag=1; //读标志位置 1
    }
}
```

定时器 0 的中断函数中，定时器装载初值，初值可以使得定时器 2ms 左右的时间就进入该中断函数一次，每进入一次 num 都加 1，值到 num 增加到 300，那么就使得 ReadTempFlag 变成 1，此时 main()函数里的 whie(1)循环里就会进入读取温度值，并进行显示到 1602 液晶屏上，具体代码如下：

```
if(ReadTempFlag==1)
        {
            ReadTempFlag=0;
            temp=ReadTemperature();
            temperature=(float)temp*0.0625;
            sprintf(displaytemp,"Temp    % 7.3f",temperature);    //打印温度值
            LCD_Write_String(0,1,displaytemp);      //显示第二行
        }
```

3.显示 1602 液晶屏的代码有专门的章节分析讲解，这里详细分析一下 ReadTemperature()温度读取函数，可以看到函数里 Init_DS18B20()初始化 18B20 温度传感器，然后使用 WriteOneChar()函数写入了 0xCC 和 0x44 这些参数，

表 3-113　DS18B20 控制命令

指　　令	约定代码	操　　作　　说　　明
读 ROM	33H	读 DS18B20 ROM 中的编码(即读 64 位地址)
ROM 匹配	55H	发出此命令后紧接着发出 64 位 ROM 编码，访问单总线上与编码相对应 DS18B20 使之做出响应，为下一步对该 DS18B20 的读写作准备
搜索 ROM	F0H	用于确定挂接在同一总线上 DS18B20 的个数和识别 64 位 ROM 地址，为操作各器件作好准备
跳过 ROM	CCH	忽略 64 位 ROM 地址，直接向 DS18B20 发温度变换命令，适用于单片机工作
警报搜索	ECH	该指令执行后，只有温度超过设定值上限或下限的片子才做出响应
温度转换	44H	启动 DS18B20 进行温度转换
读暂存器	BEH	读暂存器 9 个字节内容(读取 RAM 里前 2 个寄存器字段，前两个就是温度值)
写暂存器	4EH	将数据写入暂存器的 TH、TL 字节
复制暂存器	48H	把暂存器的 TH、TL 字节写到 E2RAM 中
重新调 E2RAM	B8H	把 E2RAM 中的 TH、TL 字节写到暂存器 TH、TL 字节
读电源供电方式	B4H	启动 DS18B20 发送电源供电方式的信号给主 CPU

首先初始化 18B20，然后写入命令 CCH 跳过读序列号的操作直接向 DS18B20 发温度变换命令，然后就发命令 44H，命令 DS18B20 内部进行温度转换；延时 10 个毫秒之后再次初始化 DS18B20 传感器，重复 CC 命令，然后再发 BE 命令即读取 RAM 的 2 个寄存器字段，前面两个就是温度值；当 DS18B20 收到这个命令后，分两次调用 ReadOneChar()函数读取一个两个 8 位的温度值，保持到变量 a 和 b 中；最后把 b 的高位左移 8 位，将 a 和 b 合并到一起组成 16 位的变量 t，最后返回 t 的值给 ReadTemperature()函数，那么 ReadTemperature()函数读取出来的就是温度值了，具体代码如下：

```
unsigned int ReadTemperature (void)
{
    unsigned char a=0;
    unsigned int b=0;
    unsigned int t=0;
    Init_DS18B20();
    WriteOneChar(0xCC);     // 跳过读序列号的操作
    WriteOneChar(0x44);     // 启动温度转换
    DelayMs(10);
    Init_DS18B20();
    WriteOneChar(0xCC);     //  跳过读序列号的操作
    WriteOneChar(0xBE);     // 读取 RAM 里前 2 个寄存器字段，前两个就是温度值
    a = ReadOneChar();       // 地位
    b = ReadOneChar();       // 高位
    b<<=8;
    t=a+b;
    return(t);
```

```
}
```
4.关于 WriteOneChar()函数是写一个字节到 DS18B20 里，ReadOneChar()是读从 DS18B20 中读取一个数据

```
void WriteOneChar(unsigned char dat)
{
	unsigned char i=0;
	for (i=8; i>0; i--)
	{
		DQ = 0;
		DQ = dat&0x01;
		DelayUs2x(25);
		DQ = 1;
		dat>>=1;
	 }
	DelayUs2x(25);
}
unsigned char ReadOneChar(void)
{
	unsigned char i=0;
	unsigned char dat = 0;
	for (i=8;i>0;i--)
	{
		DQ = 0;	//拉低总线，启动输入
		dat>>=1;	//移出 1 个空白位置，准备读取数据
		DQ = 1;	//给脉冲信号
		if(DQ)
			dat|=0x80;
		DelayUs2x(25); //释放总线
	}
	return(dat);
}
```
程序分析：

向 DS18B20 写入数据，在单总线上每个时序只传送一位数据，并且分为写 0 的过程和写 1 的过程，因为只有一根数据线要分别实现写 0 或者写 1，所以对时序的要求是非常有讲究的，下面进行进一步的详细分析：

1） 写 0 的过程，首先拉低总线 60us，然后抬高总线 5us，完成写 0 的动作，其中 60us 的前半部分中包括了写 0 的这个操作。
2） 写 1 的过程：拉低总线 5us，然后抬高总线 60us 完成，其中前 5us 是做准备，后面的 60us 的前半部分表示对数据线写 1.
3） 读 0 和 1 的过程比较相同：拉低总线 5us，然后释放总线，读取总线，如果为 0 则读入 0；如果为 1 则读入 1，具体实现可以参看代码。

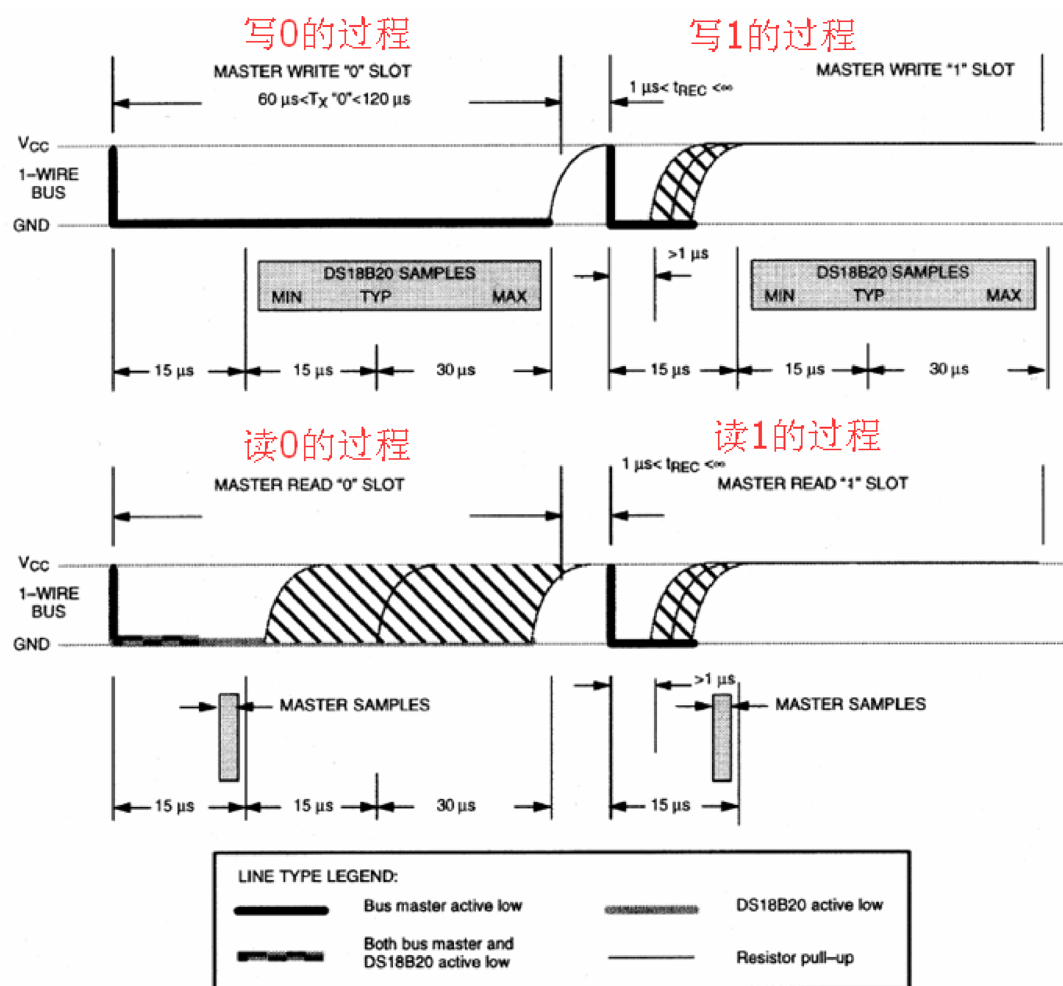


图 3-163 读/写时序流程图

3.24.11 更多有关DS18B20 温度传感器的例程

更多 DS18B20 温度传感器相应的例程可通过北京航空航天大学出版社下载专区下载，如下表 3-115：

表 3-11518B20 温度传感器更多丰富例程介绍（含详细注释和文档分析）

序号	例程功能
例程 01	18B20 初始化程序
例程 02	18B20 温度采集在 1602 液晶显示屏上显示
例程 03	18B20 多个温度采集液晶显示
例程 04	18B20 温度可调上下限 1602 屏显示

3.25 直流电机

3.25.1 直流电机的介绍

直流电机是指能将直流电能转换成机械能（直流电动机）或将机械能转换成直流电能（直流发电机）的旋转电机。它是能实现直流电能和机械能互相转换的电机。

3.25.2 直流电机的内部结构

直流电机的结构应由定子和转子两大部分组成。1）直流电机运行时静止不动的部分称为定子，定子的主要作用是产生磁场，由机座、主磁极、换向极、端盖、轴承和电刷装置等组成。2）运行时转动的部分称为转子，其主要作用是产生电磁转矩和感应电动势，是直流电机进行能量转换的枢纽，所以通常又称为电枢，由转轴、电枢铁心、电枢绕组、换向器和风扇等组成。如图 3-164。下面我们详细了解下：

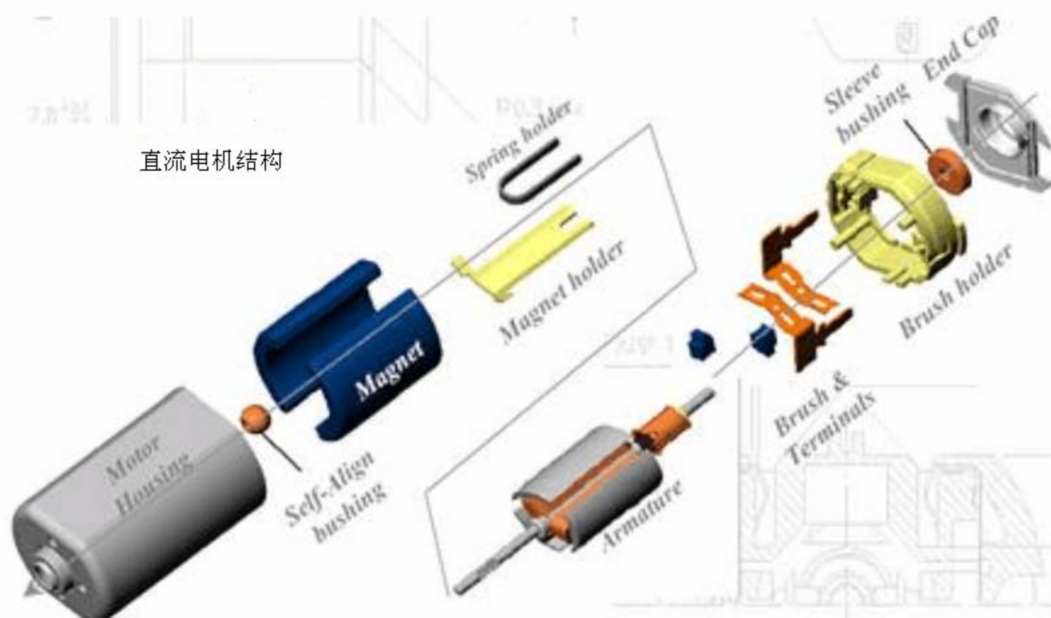


图 3-164 直流电机内部结构

1）电机铁壳组件(定子组件)

A．铁壳—一个用以包裹磁铁及轴承并被固定于端盖上的金属外壳。

B．轴承—支持属于电枢组件的轴枝，它为轴枝提供润滑作用以减少磨损。

C．磁铁—永久磁铁是产生磁场的关键元件，此外，它还为电机制造转矩及旋转力。

D．弹弓架—用以将磁铁固定至铁壳中

2）电枢组件(转子组件)

A．换向器—换向器片通常为铜材料制作，它通过旋转及与电刷滑动接触，此元件得以转换电流的方向。
B．叠片铁心 —流过电流的漆包线绕组放在槽中。
C．漆包线—构成电枢绕组。
D．轴枝—电机转动就是轴枝的旋转

3）端盖组件
A．电刷架—用来固定电刷并与铁壳绝缘。
B．端盖—是金属冲压件，用以固定轴枝
C．接线端—通常是成对的，是连接电机的电力输入通道。
D．电刷—通常以碳做为材料，在电枢组件旋转时，电刷与换向器滑动接触从而使电流得以从接线端一端流向电枢。
E．电刷片—用以保持电刷位置，使电刷得以滑动并以适当的压力与换向器相连接

3.25.5 直流电机内部运行原理

通电的导体在磁场中会受力而产生运动，这是物理学里的一个原理，这里简单的介绍一下，我们使用左手定则：首先要知道磁铁的感应线方向（N 级到 S 级）,然后伸出左手,手心与感应线垂直,（也就是手心对着 N 级）大拇指与食指成 90 度，四指指向电流方向,大拇指指的就是导体运动方向，如下图 3-165 所示

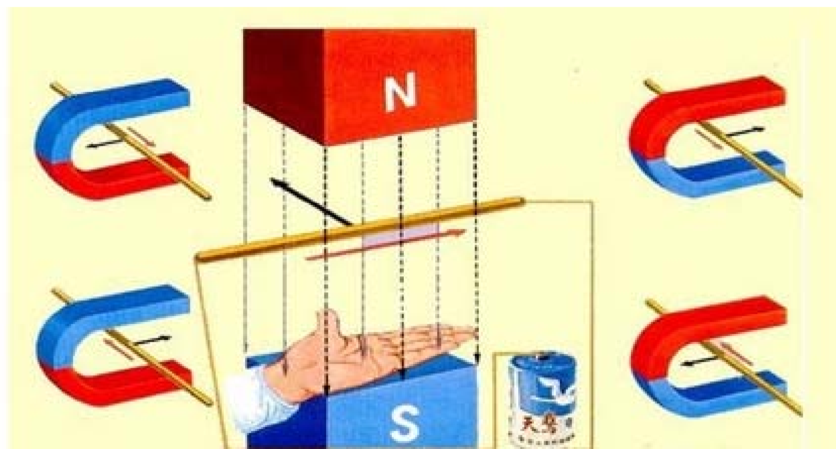


图 3-165 磁场受力导体运动方向

直流电机就是利用了这个原理来产生转动的，下图 3-166 为直流电机内部结构图，在图中，线圈连着换向片，换向片固定于转轴上，随电机一起旋转，换向片之间及换向片与转轴之间均互相绝缘，它们构成的整体称为换向器。电刷 A、B 在空间上固定不动。

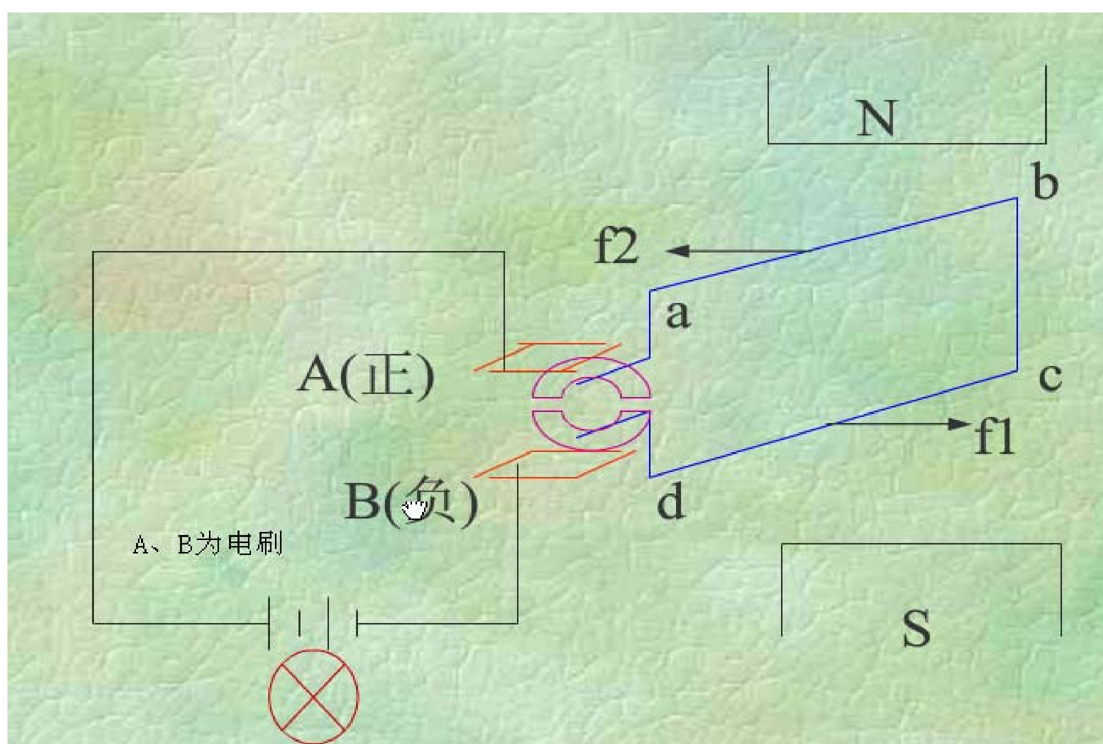


图 3-166 直流电机内部结构图

在电机的两电刷端加上直流电压，由于电刷和换向器的作用将电能引入电刷线圈中，并保证了同一个级下线圈边的电流始终是一个方向，继而保证了该极下线圈边所受的电磁力方向不变，保证了电动机能连续地旋转，以实现将电能转换为机械能以拖动生产机械，图的中 f1、f2 为导体的受力方向，可以通过上面的方法得出，所以在电刷上电的时候就能按照力的方向旋转起来。

3.25.6 单片机如何控制直流电机

举个例：我们放一盆水，一直不停的放要 1 分钟放满，但我为了控制放满的时间，在每一秒的时间里需要开一下，关一下。而这开和关的时间比值就可以认为是脉冲的占空比，开的时间长，相应的关的时间就会缩短（每秒必须完成一次开和关，相当于脉冲的频率）。而放满的时间就可以通过这样的方式来调节（相当于控制输出）这样通过调整开和关的时间（脉冲宽度）来调整输出的，就是脉宽调制。

利用脉宽调制(PWM)方式实现调光/调速的好处是电源的能量能得到充分利用，电路的效率高，如下图。例如：当输出为 50%的方波时，脉宽调制(PWM)电路消耗的电源能量也为 50%，即几乎所有的能量都转换为负载功率输出。而采用常见的电阻降压调速时，要使负载获得电源最大输出功率 50%的功率，电源必须提供 71%以上的输出功率，这其中 21%消耗在电阻的压降及热耗上。有时电路的转换效率是非常重要的。

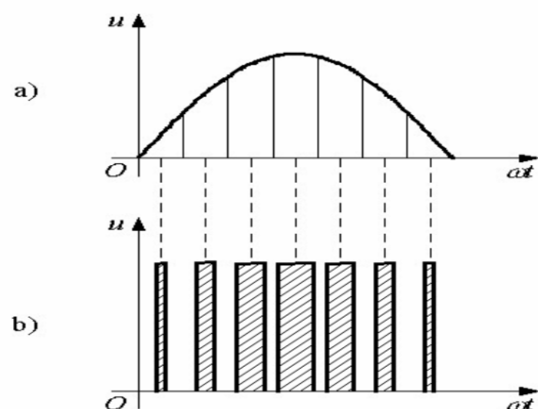


图 3-167 单片机输出脉宽调制(PWM)

直流电机的旋转速度如何控制呢？将直流脉冲序列（PWM 波）加到直流电机两端，通过单片机输出脉宽调制(PWM)直流电动机控制器来实现控制，改变脉冲的占空比而达到改变电机两端电压，使得负载上的平均接通时间从 0-100%变化，从而控制电机转速，以达到调整速度的目的；脉冲宽度越大即占空比越大，提供给电机的平均电压越大，电机转速就高。反之脉冲宽度越小，则占空比越越小。提供给电机的平均电压越小，电机转速就低。

3.25.7 为什么电机需要专用驱动芯片

UNL2003 芯片可以增加驱动力，并且还能充当隔离，防止电机产生的冲击电流回流烧坏 MCU 主控 IC。

如下图 3-168，左边是 UNL2003 芯片以及内部结构，里面有 7 对达林顿管；右边是其中一对达林顿管的内部等效电路图；可以看到 UNL2003 的每一对达林顿都串联一个 2.7K 的基极电阻，在 5V 的工作电压下它能与 TTL 和 CMOS 电路直接相连，UNL2003 芯片内部框架图及单个达林顿晶体管的等效电路图功能：

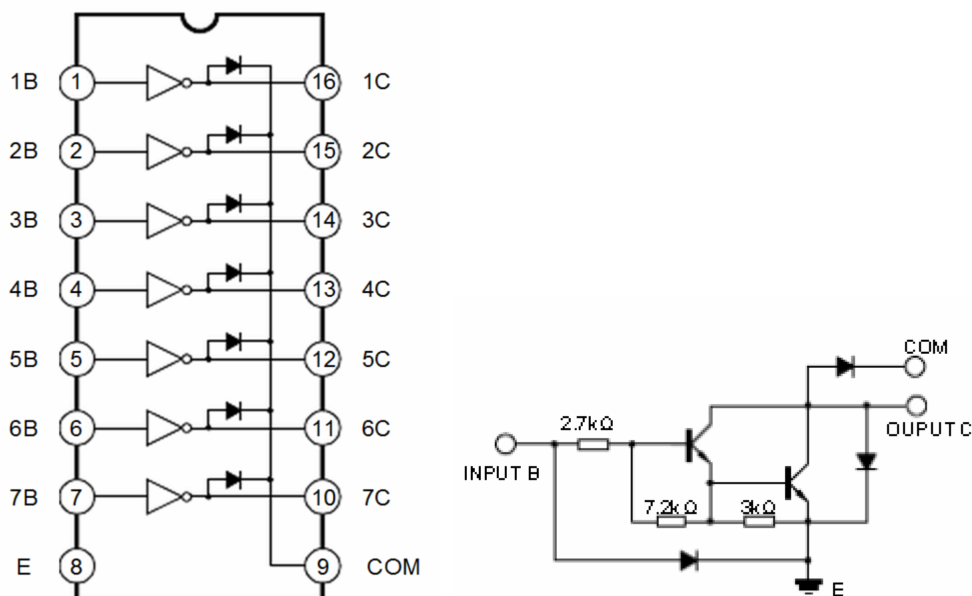


图 3-168 UNL2003 芯片的管脚图及等效电路图功能

ULN2003 由 7 组达林顿晶体管阵列和相应的电阻网络以及钳位二极管网络构成，具有同时驱动 7 组负载的能力，即可以并联使用，在相应的输出管脚上串联几个欧姆的均流电阻后再并联使用，防止阵列电流不平衡。

ULN2003 芯片的输出结构是集电极开路的，也就是说回灌输入；如果电流驱动不够还可以把 COM 端芯片的第 9 脚 COM 引脚公共端与负载相连，在输出端接一个上拉电阻，在驱动负载的时候，电流是由电源通过负载灌入 ULN2003 的。

ULN2003，有时接电源是为了对输出负载增加驱动力，所有输出脚与该引脚之间串有一个二极管，称为钳位二极管，如果输出因某种原因导致电压超出 VCC 及该二极管的压降之和，该二极管将导通使输出电压限定，从而保护了器件。

3.25.8 硬件原理图与连接

硬件连接原理图如下图 3-169 所示：

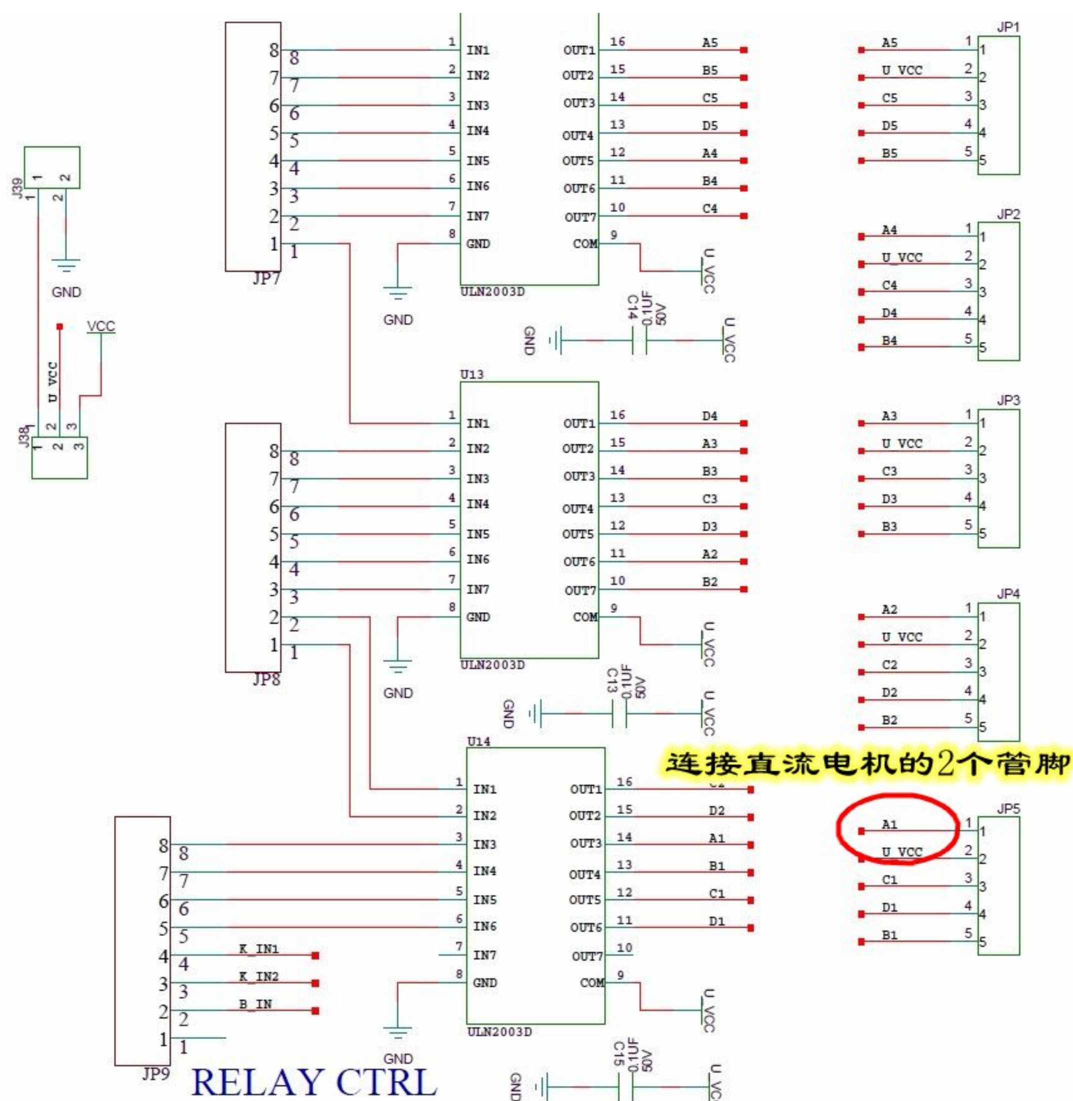


图 3-169 硬件连接原理图

将 J38 短接 VCC 端，J17 短接 P10 端，将直流电机插头插入 JP5 插座的 1、2 脚（A1 和 VCC）；再用一根 8 位排线将板上的 JP9 插针的与单片机 P1 口 JP13 插针连接；以下例程均

使用第一路步进电机为例，其中硬件连接按此方法连接，如下图 3-170：

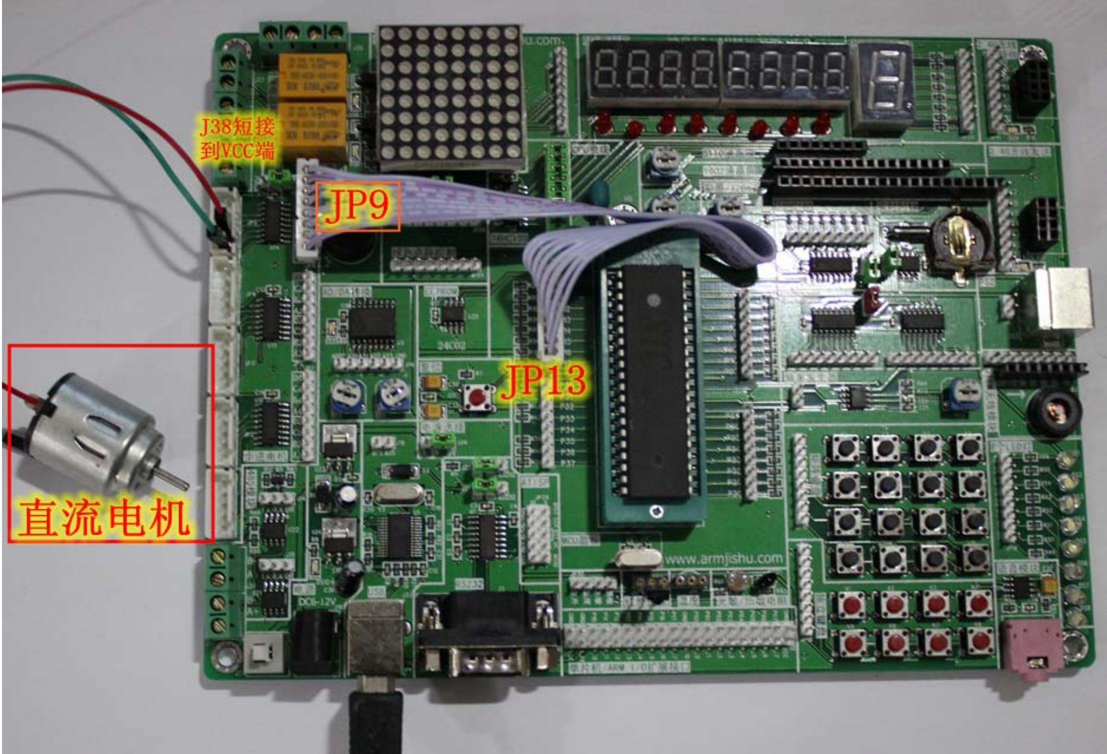


图 3-170 硬件连接实物图

硬件连接对应关系如下表 3-116 所示：

表 3-116 硬件连接对应关系

<table>
<tr><th>单片机接口</th><th>插座 1</th><th>方式</th><th>插座 2</th><th>线缆</th><th>功能</th></tr>
<tr><td>P1</td><td>JP13（A 向左）</td><td>直连</td><td>JP9（A 向右）</td><td>1 根 8 位排线</td><td>单片机控制步进电机</td></tr>
<tr><td colspan="6">将 J38 短接 VCC 端，将步进电机插头插入 JP5 插座 1、2 脚（A1 和 VCC）
注意：下载程序前请先断开排线，下载完后再插上排线</td></tr>
</table>

3.25.9 例程 01 直流电机恒速转动

代码如下：

```
/*******************************************************
*  例程：直流电机恒速转动
*  作者：www.armjishu.com
*  版本：v1.0
*  内容：直流电机恒速转动
******************************************************/
#include<reg52.h>
                //包含头文件，一般情况不需要改动，头文件包含特殊功能寄存器的定义
unsigned char timer1;
sbit PWM=P1^7;
```

```
/*------------------------------------------------
                初始化函数
------------------------------------------------*/
void system_Ini() //系统初始化
{
    TMOD|= 0x10;   //使用模式 1，16 位定时器，使用"|"符号可以在使用多个定时器时不受影响
    TH1 = 0xfe;     //给定时器 1 初始值 0xfe0c.每个机器周期大约为 1us;
    TL1 = 0x0c;     //定时器最大值从 0 开始计数一直到 0xffff（等于 65535）溢出.
    //定时初值应为 65536－500＝65036，将 65036 化为十六进制即为 0xfe0c.
    TR1  = 1;      //定时器 1 开关打开
}
/*------------------------------------------------
                    主函数
------------------------------------------------*/
 main()
{
    timer1=0;
    system_Ini(); //系统初始化
    while(1)        //主循环
    {
        if(timer1>10000)
        {
            timer1=0;
        }
        if(timer1<1)
        {
            PWM=0;    //PWM 为低电平
        }
        else
        {
            PWM=1;    //PWM 为高电平
        }
    }
}
/*------------------------------------------------------------------------
                      中断函数
 [ t1 (0.5ms)中断]  中断中做  PWM  输出
  ------------1000/(0.02ms*250)=200Hz
  1  定时器 0 的中断号，0  外部中断 1 ，2  外部中断 2 ，
  3  为定时器 1 的中断号 ，4  串口中断
------------------------------------------------------------------------*/
void T1zd(void) interrupt 3   //3  为定时器 1 的中断号
{
```

```
        TH1 = 0xfe;     //给定时器 1 初始值 0xfe0c.每个机器周期大约为 1us;
        TL1 = 0x0c;     //定时器最大值从 0 开始计数一直到 0xffff（等于 65535）溢出.
        //定时初值应为 65536－500＝65036，将 65036 化为十六进制即为 0xfe0c.
         timer1++;
    }
```

硬件连接对应关系如下表 3-116 所示：

表 3-116 硬件连接对应关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P1	JP13（A 向左）	直连	JP9（A 向右）	1 根 8 位排线	单片机控制步进电机
将 J38 短接 VCC 端，将直流电机插头插入 JP5 插座 1、2 脚（A1 和 VCC） 注意：下载程序前请先断开排线，下载完后再插上排线					
实验现象：下载程序后，我们可以看到直流电机转动					

3.26 步进电机

3.26.1 什么是步进电机

步进电机是机电控制中一种常用的组件，在自动化仪表、自动控制、机器人、自动生产流水线等领域的应用相当广泛。它的用途是将电脉冲转化为角位移，通俗的说，当步进驱动器接收到一个脉冲信号，它就驱动步进电机按设定的方向转动一个固定的角度(步进角)。

通过控制秒冲个数即可控制角位移量，可以实现准确定位的目的；同时通过控制脉冲频率来控制电机转动的速度和加速度，从而达到调速的目的，即给电机加一个脉冲信号，电机则转过一个步距角。

3.26.4 步进电机是怎样转动起来的

步进电机是一种将电脉冲转化为角位移的执行机构。通俗一点讲：当步进驱动器接收到一个脉冲信号，它就驱动步进电机按设定的方向转动一个固定的角度（及步进角），它的旋转是以固定的角度一步一步运行的。您可以通过控制脉冲个数来控制角位移量，从而达到准确定位的目的；同时您可以通过控制脉冲频率来控制电机转动的速度和加速度，从而达到调速的目的。由于步进电机的精度高，误差不累积，由此它通常在数控设备中担负精确传动的作用。

下面我们以四相步进电机为例子简要说明步进电机是怎样转动起来。四相步进电机，采用单极性直流电源供电。只要对步进电机的各相绕组按合适的时序通电，就能使步进电机步进转动。下图 3-171 为四相步进电机工作原理示意图：

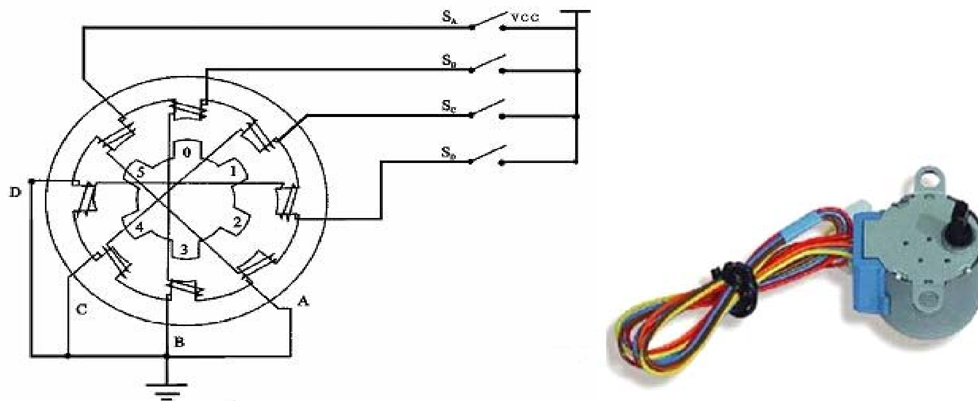


图 3-171 四相步进电机步进示意图

假设开始时，开关 SB 接通电源，SA、SC、SD 断开，B 相磁极和转子 0、3 号齿对齐，同时，转子的 1、4 号齿就和 C、D 相绕组磁极产生错齿，2、5 号齿就和 D、A 相绕组磁极产生错齿。

当开关 SC 接通电源，SB、SA、SD 断开时，由于 C 相绕组的磁力线和 1、4 号齿之间磁力线的作用，使转子转动，1、4 号齿和 C 相绕组的磁极对齐。而 0、3 号齿和 A、B 相绕组产生错齿，2、5 号齿就和 A、D 相绕组磁极产生错齿。依次类推，A、B、C、D 四相绕组轮流供电，则转子会沿着 A、B、C、D 方向转动。

3.26.2 步进电机和普通直流电机的区别

步进电机每给一次完整的信号就步进一次，也就是给一个脉冲，就会转过固定的角度，它是靠磁场磁铁吸引；而直流电机通电就转，它的速度取决于占空比，它是直接把电能转化成电机的动能，它能最大发挥强大运转动力来旋转，但它的缺点是转多少角度、精确转动定位这些是很难可控的;所以说精准化运转是步进电机的优势,而强大动力是直流电机的优势。

常见的步进电机步进角度是 5 度，需要步进 72 次才能转一圈，精确的转动在一些数控领域或者车床控制领域就显得尤其重要了，比如需要在某个坐标钻孔，如果使用直流电机，则很难精确找到坐标，而如果使用步进电机，则可以将坐标转换为步进次数。

因此,单片机在控制直流电机和步进电机时,直流电机是需要给高电平或者低电平即可，步进电机需要给相控信号才能驱动。当然在很多领域，定位的精度还是来源于驱动器；设计一个好的驱动器加上编码器，不管是无刷、有刷、交流电机、还是步进电机，随便找个电机也能精确定位的，更深入的内容请关注光盘资料。

3.26.6 硬件原理与连接

硬件连接的原理图如下图 3-174 所示：

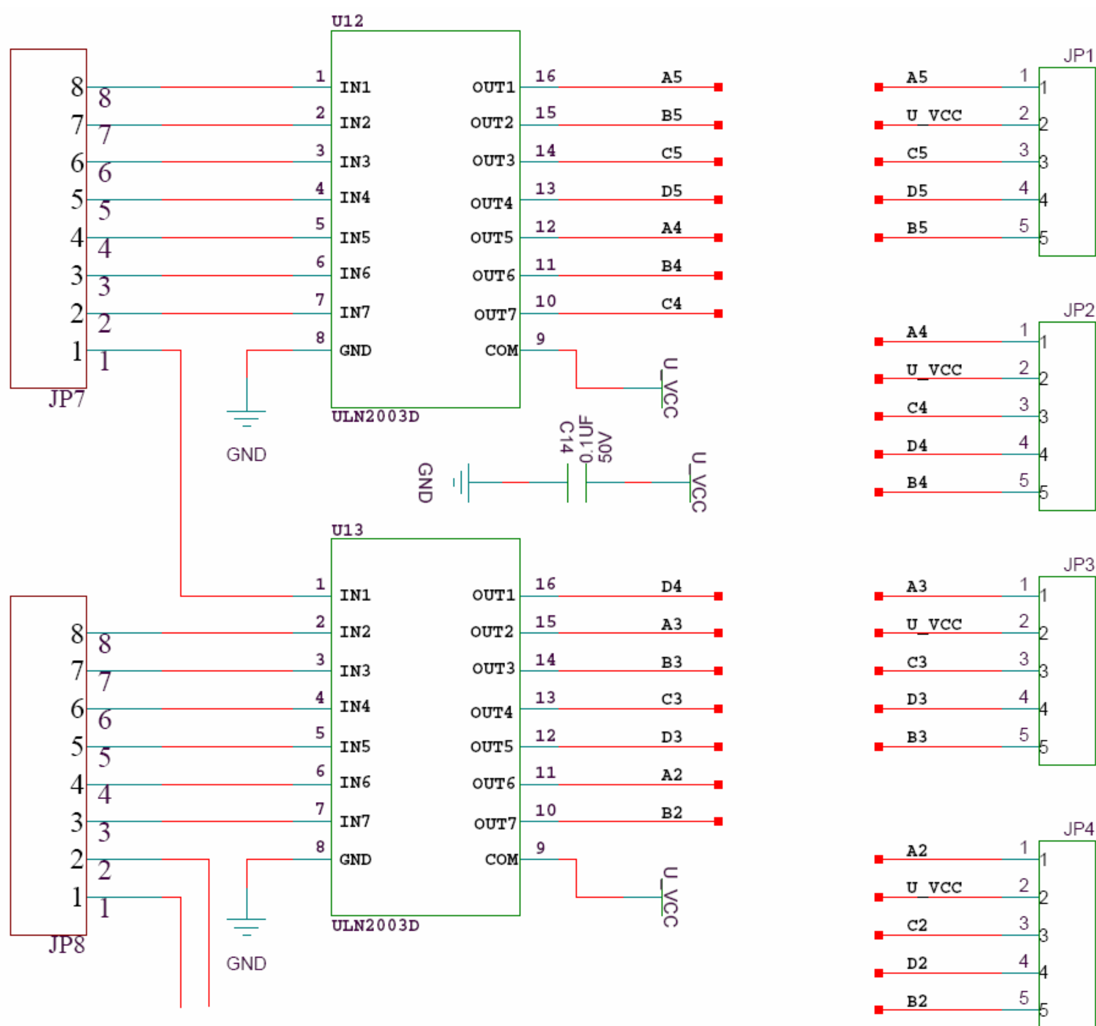


图 3-174 硬件连接原理图

将 J38 短接 VCC 端，J17 短接 P10 端，将步进电机插头插入 JP5 插座；再用一根 8 位排线将板上的 JP9 插针的与单片机 P1 口 JP13 插针连接；用一根 8 位排线将板上的 JP23 插针的与单片机 P0 口 JP15 插针连接;用一根 8 位排线将板上的 JP22 插针的与单片机 P2 口 JP16 插针连接；最后用 4 根单针杜邦线将单片机的 P3.2 与 JP18 第 1 脚连接、 P3.3 与 JP18 第 2 脚连接、P3.4 与 JP18 第 3 脚连接、P3.5 与 JP18 第 4 脚连接；以下例程均使用第一路步进电机为例，其中硬件连接按此方法连接，如下图 3-176：

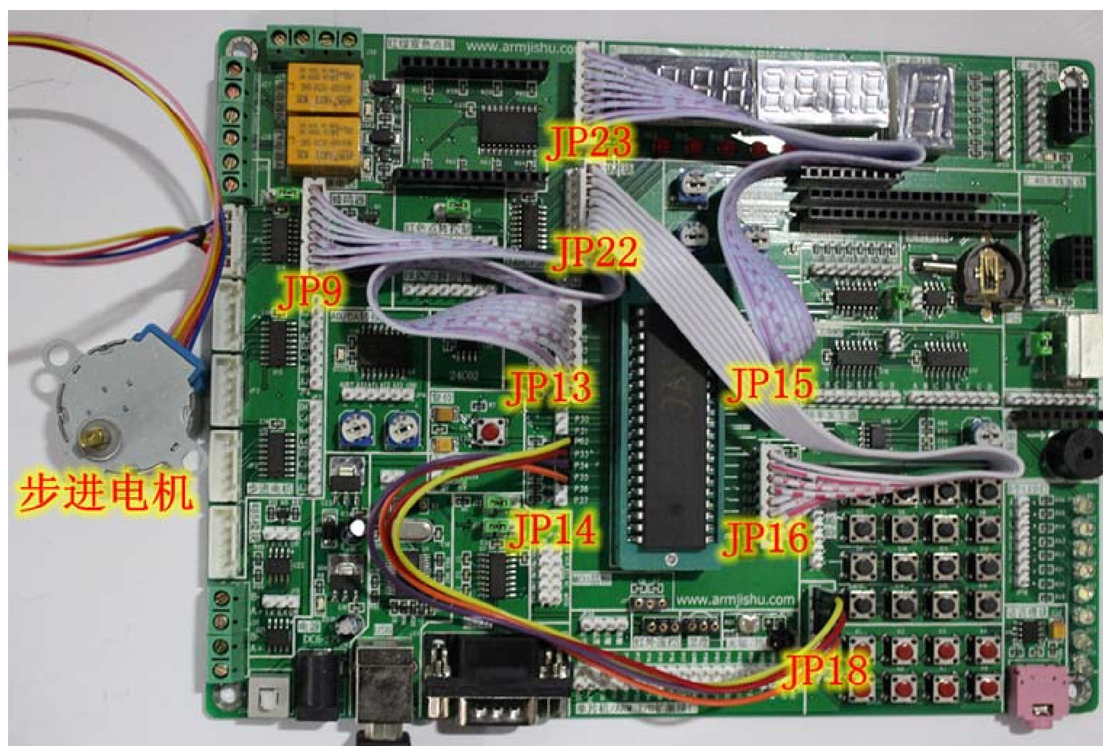


图 3-176 硬件连接实物图

硬件连接关系如下表 3-117 所示：

表 3-117 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P1	JP13（A 向左）	直连	JP9（A 向右）	1 根 8 位排线	单片机控制步进电机
P0	JP15（A 向左）	直连	JP23(A 向右)	1 根 8 位排线	单片机控制数码管的段
P2	JP16（A 向右）	直连	JP22(A 向右)	1 根 8 位排线	单片机控制数码管的位
P3.2	JP14 第 3 脚	直连	JP18 第 1 脚	单根杜邦线	单片机接收按键信号 1
P3.3	JP14 第 4 脚	直连	JP18 第 2 脚	单根杜邦线	单片机接收按键信号 2
P3.4	JP14 第 5 脚	直连	JP18 第 3 脚	单根杜邦线	单片机接收按键信号 3
P3.5	JP14 第 6 脚	直连	JP18 第 4 脚	单根杜邦线	单片机接收按键信号 4
将 J38 短接 VCC 端，J17 短接 P10 端，最后将步进电机插头插入 JP5 插座					

步进电机驱动方式：

1） 1 相励磁法：每一瞬间只有一个线圈导通，其他线圈休息。其特点是励磁方法简单，耗电低，精确度良好。但是力矩小、震动大，每次励磁信号走的角度是标称角度。

2） 2 相励磁法：每一瞬间有两个线圈同时导通，特点是力矩大、震动较小，每次励磁转动角度是标称角度。

3） 1-2 相励磁法：1 相和 2 相轮流交替导通，精度较高，且运转平滑。每送一个励磁信号转动二分之一标称角度。有称为半步驱动。4 相电机中，1、2 种方式称 4 相 4 拍，3 种称 4 相 8 拍。

1 相励磁、2 相励磁和 1-2 相励磁方式如下图 3-177 所示：

1 相励磁						2 相励磁						1-2 相励磁				
步	A	B	C	D		步	A	B	C	D		步	A	B	C	D
1	1	0	0	0		1	1	1	0	0		1	1	0	0	0
2	0	1	0	0		2	0	1	1	0		2	1	1	0	0
3	0	0	1	0		3	0	0	1	1		3	0	1	0	0
4	0	0	0	1		4	1	0	0	1		4	0	1	1	0
5	1	0	0	0		5	1	1	0	0		5	0	0	1	0
6	0	1	0	0		6	0	1	1	0		6	0	0	1	1
7	0	0	1	0		7	0	0	1	1		7	0	0	0	1
8	0	0	0	1		8	1	0	0	1		8	1	0	0	1

图 3-177 1 相励磁、2 相励磁和 1-2 相励磁方式

3.26.7 例程 01 步进电机转动原理 1

代码如下：

```
/*******************************************************
* 例程：步进电机转动原理 1
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：本程序用于测试 4 相步进电机常规驱动；使用 1 相励磁。
*******************************************************/
#include <reg52.h>//包含头文件，一般情况不需要改动，
                  //头文件包含特殊功能寄存器的定义
sbit D1=P1^4;   //定义步进电机连接端口
sbit C1=P1^5;   //定义步进电机连接端口
sbit B1=P1^6;   //定义步进电机连接端口
sbit A1=P1^7;   //定义步进电机连接端口
#define Coil_A   {A1=1;B1=0;C1=0;D1=0;}   //A 相通电，其他相断电
#define Coil_B   {A1=0;B1=1;C1=0;D1=0;}   //B 相通电，其他相断电
#define Coil_C   {A1=0;B1=0;C1=1;D1=0;}   //C 相通电，其他相断电
#define Coil_D   {A1=0;B1=0;C1=0;D1=1;}   //D 相通电，其他相断电
#define Coil_OFF {A1=0;B1=0;C1=0;D1=0;}   //全部断电
unsigned char Speed;
/*------------------------------------------------
 uS 延时函数，含有输入参数 unsigned char t，无返回值
 unsigned char 是定义无符号字符变量，其值的范围是
 0~255 这里使用晶振 12M，精确延时请使用汇编,大致延时
 长度如下 T=tx2+5 uS
------------------------------------------------*/
void DelayUs2x(unsigned char t)
{
 while(--t);
}
/*------------------------------------------------
```

```
 mS 延时函数，含有输入参数 unsigned char t，无返回值
 unsigned char 是定义无符号字符变量，其值的范围是
 0~255 这里使用晶振 12M，精确延时请使用汇编
------------------------------------------------*/
void DelayMs(unsigned char t)
{
 while(t--)
 {
     //大致延时 1mS
     DelayUs2x(245);
     DelayUs2x(245);
 }
}
/*------------------------------------------------
                    主函数
------------------------------------------------*/
main()
{
 Speed=50; //调整转动速度，速度不可以调节的过快，不然就没有力矩转动了
 while(1)   //主循环(while() 是单片机的一种基本循环模式。
                   //当满足条件时进入循环，不满足跳出)
   {
     Coil_A                    //遇到 Coil_A   用{A1=1;B1=0;C1=0;D1=0;}代替
     DelayMs(Speed);           //改变这个参数可以调整电机转速 ,
                               //数字越小，转速越大,力矩越小
     Coil_B                    //遇到 Coil_B   用{A1=0;B1=1;C1=0;D1=0;}代替
     DelayMs(Speed);
     Coil_C                    //遇到 Coil_C   用{A1=0;B1=0;C1=1;D1=0;}代替
     DelayMs(Speed);
     Coil_D                    //遇到 Coil_D   用{A1=0;B1=0;C1=0;D1=1;}代替
     DelayMs(Speed);
   }
}
```

硬件连接关系如下表 3-118 所示：

表 3-118 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P1	JP13（A 向左）	直连	JP9（A 向右）	1 根 8 位排线	单片机控制步进电机
将 J38 短接 VCC 端，将步进电机插头插入 JP5 插座					
实验现象：下载程序后，我们可以看到步进电机转动					

知识要点：

1. 一相励磁法：每一瞬间只有一个线圈导通，其他线圈休息。其特点是励磁方法简单，

耗电低，精确度良好。但是力矩小、震动大，每次励磁信号走的角度是标称角度。

2. 速度不可以调节的过快，不然就没有力矩转动了。

3.26.8 更多有关步进电机的例程

更多步进电机相应的例程可通过北京航空航天大学出版社下载专区下载，如下表 3-119：

表 3-119 步进电机更多丰富例程介绍（含详细注释和文档分析）

序号	例程功能
例程 01	步进电机转动原理 1
例程 02	步进电机转动原理 2
例程 03	步进电机转动原理 3
例程 04	步进电机正反转
例程 05	步进电机按键控制
例程 06	步进电机转速数码管显示
例程 07	步进电机调速原理
例程 08	步进电机综合控制

3.27 继电器

3.27.1 继电器的简介

在人类的生活中到处可以遇到小力量控制大力量的产品；例如电梯；例如开车时踩刹车汽车就会开得更快；例如按下某个开关，洗衣机就会完成自动洗衣的全过程；同样的道理，人直接接触高电压设备会非常危险，如果可以通过工具来间接控制就会安全很多，继电器就具有这个功能，在电路通常采用继电器进行控制大电压大电流的设备。

继电器是具有控制和隔离功能的自动开关元件。例如用 5V 电源控制 220V 的电压设备，最简单的方式就是通过继电器来操作；5V 通电可以使得电磁吸合，电磁吸合来控制另一端的正负极闭合或断开，从而控制 220V 或者甚至更高的电压的开合操作；由于继电器通过中间的电磁装置进行隔离，所以这样可以最大限度的保证操作安全。

继电器是一种电子控制器件，它具有控制系统（又称输入回路）和被控制系统（又称输出回路），同城应用于自动控制电路中，它实际上是用较小的电流去控制较大电流的一种自动开关。故在电路中起着自动调节、安全保护、转换电路等作用；继电器一般可以分为电磁式继电器、固态继电器等；除了上述功能之外，继电器的作用还有很多，它可以与其他电器一起组成程序控制线路，实现自动化运行，在此同时，它还将信号综合起来，可以把经过继电器的多个控制信号综合，达到预定的控制效果。

3.27.2 单个电磁继电器的工作原理

电磁继电器原理可以简单理解为就是一个电磁铁，这个电磁铁的衔铁可以闭合或断开一个或数个接触点；当电磁铁的绕组中有电流通过时，衔铁被电磁铁吸引，因而就改变了触点

的状态；单片机是一个弱电器件,一般情况下它们大都工作在 5V 甚至更低.驱动电流在 mA 级以下，而要把它用于一些大功率场合，比如控制电动机，显然是不行的。所以，就要有一个环节来衔接，这个环节就是所谓的"功率驱动"。继电器驱动就是一个典型的、简单的功率驱动环节。在这里，继电器驱动含有两个意思：一是对继电器进行驱动,因为继电器本身对于单片机来说就是一个功率器件；还有就是继电器去驱动其他负载，比如继电器可以驱动中间继电器，可以直接驱动接触器，所以继电器驱动就是单片机与其他大功率负载接口。这个很重要，因为很多初学者迷惑不解的是：一个小小的芯片，怎么会有如此强大的威力来控制像电动机这样强大的东西？继电器输入两端是小电流，而输出端连接的是大电流，中间是靠电磁铁来隔离输入和输出两端的，电磁继电器的结构图如下图 3-178：

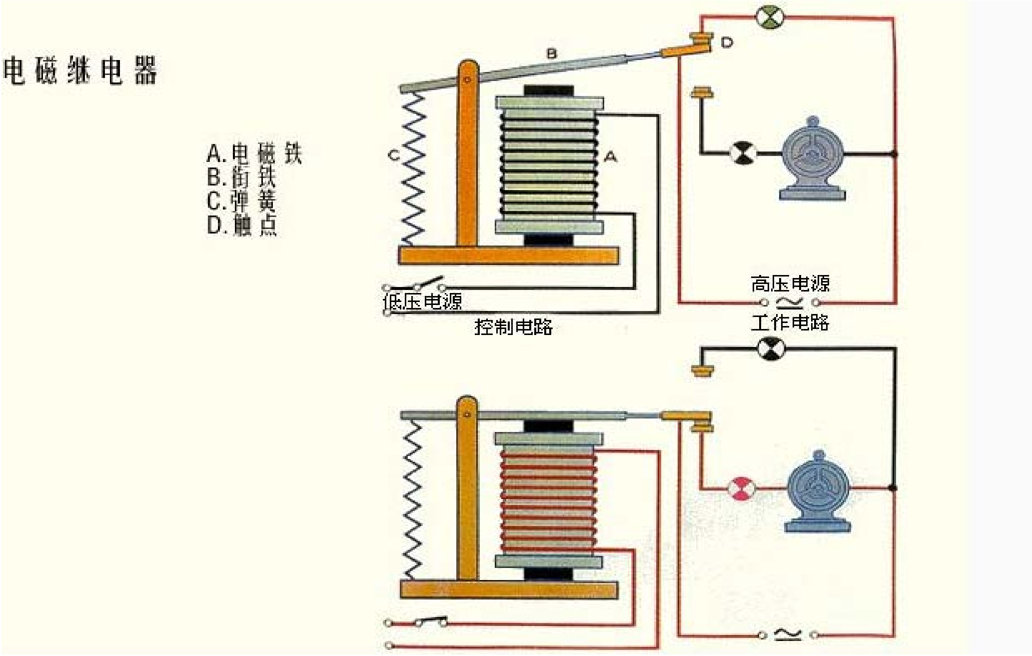


图 3-178 电磁继电器的结构图

电磁继电器是利用电磁铁铁芯与衔铁间产生吸力作用来产生吸合和断开操作的继电器，在线圈两端（低电源控制端）加上一定的电压，线圈中就会流过一定的电流，从而产生电磁效应，衔铁 B 就会在电磁力吸引的作用下克服返回弹簧的拉力吸向铁芯，从而带动衔铁的动触点与静触点（常开触点）吸合。当线圈断电后，电磁的吸力也随之消失，衔铁就会在弹簧的反作用力返回原来的位置，使动触点与原来的静触点（常闭触点）吸合。这样吸合、释放，从而达到了控制高压电路导通、切断的目的。

对于继电器的“常开、常闭”触点，可以这样来区分：继电器线圈未通电时处于断开状态的静触点，称为“常开触点”；处于接通状态的静触点称为“常闭触点”。

3.27.2 继电器使用特性

1.线圈使用电压

线圈使用电压即加到线圈引出端之间的电压，最好是按额定电压选择，如果电压低于额定电压可能会影响继电器的工作；如高于工作电压也会影响产品性能，过高的电压会使得线圈温升过高，特别是在高温下，温升过高会使绝缘材料受到损伤，也会影响到继电器的工作安全。

2.瞬态电流

继电器线圈断电瞬间，线圈上可产生高于线圈额定工作电压值 30 倍以上的反峰电压，

对电子线路有极大的危害，通常采用并联瞬态抑制二极管或电阻的方法加以抑制，降低反峰电压。

3.多个继电器的并联和串联供电

多个继电器并联供电时，反峰电压高(即电感大)的继电器会向反峰电压低的继电器放电，其释放时间会延长，因此最好每个继电器分别控制后再并联才能消除相互影响；另外，不同线圈电阻和功耗的继电器不要串联供电使用，否则串联回路中线圈电流大的继电器不能可靠工作。只有同规格型号的继电器可以串联供电。

4.触点负载

加到触点上的负载应符合触点的额定负载和性质，不符合的往往容易出现问题；例如，只适合直流负载的产品不应用于交流场合；能可靠切换 10A 负载的继电器在低电平电路下不一定能可靠工作；能切换单相交流电源的继电器不一定适合切换两个不同步的单相交流负载；只规定切换交流 60HZ 的产品不应用来切换 200HZ 的交流负载。

5.切换速率

继电器切换速率应不高于其 10 倍动作时间和释放时间之和的倒数(次/s)，否则继电器触点不能稳定接通。

3.27.3 继电器种类

继电器的种类比较多，现在按作用原理进行分类：

1.电磁继电器

在输入电路内电流的作用下，由机械部件的相对运动产生预定响应的一种继电器；它包括直流电磁继电器、交流电磁继电器、舌簧继电器，节能功率继电器等。

2.固态继电器

输入输出功能由电子元件完成而无机械运动部件的一种继电器。

3.时间继电器

当加上或除去输入信号时，输出部分需延时或限时到规定的时间才闭合或断开其被控线路的继电器。

4.温度继电器

当外界温度达到规定值时而动作的继电器。

5.风速继电器

当风的速度得到一定值时，被控电路将接通或断开。

6.加速度继电器

当运动物体的加速度达到规定值时，被控电路将接通或断开。

7.其他类型继电器

如光继电器、声继电器、热继电器等。

3.27.3 硬件原理

单片机管脚控制三极管 Q2，通过三极管 Q2 来控制继电器 RK1 的工作，另外还有一个二极管 D1 起保护作用，继电器也是属于感性器件，所以不能用单片机的 I/O 口直接来控制，要在三极管等控制器件上加反相保护电路。一般实验中都是单片机通过一个 PNP 型三极管，把三极管作为电子开关来驱动继电器，继电器的开和关完全由三极管的基极电平进行控制。当三极管基极为高电平，PNP 型三极管截止，这时继电器不工作；反之为低电平

的话，PNP 型三极管导通，继电器得电吸合硬件连接原理图如下图 3-179 所示：

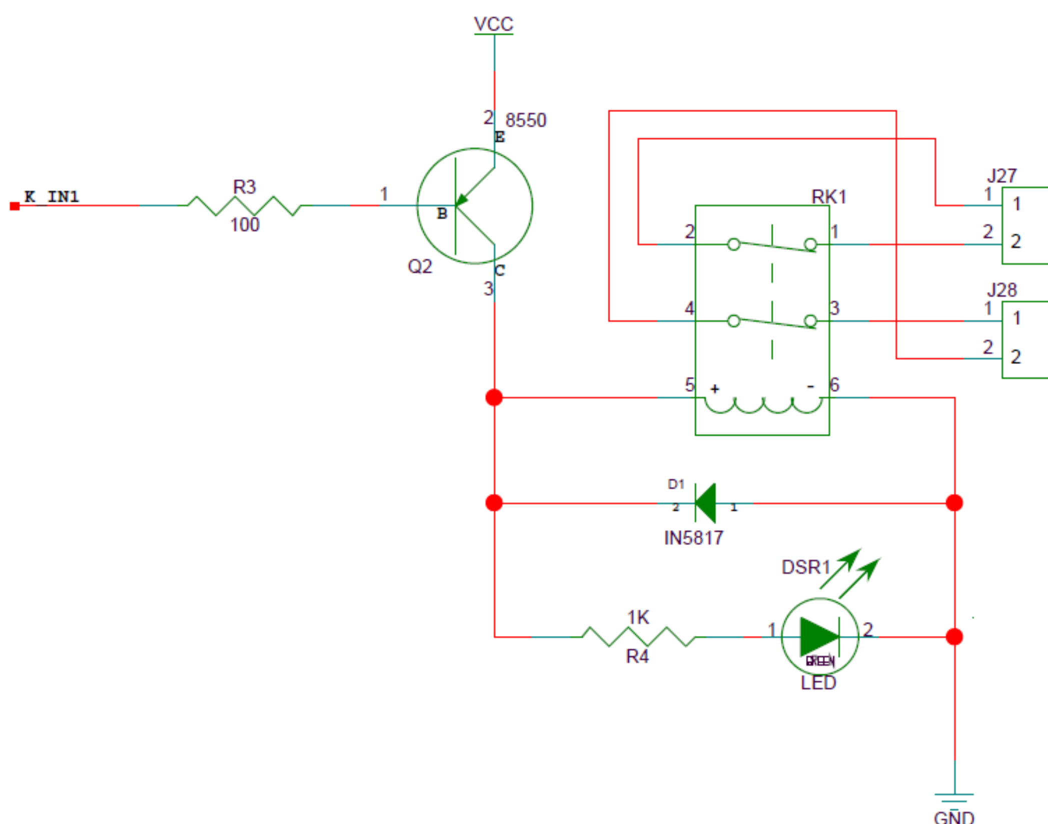


图 3-179 硬件连接原理图

简单的来说三极管有两个作用一个是放大作用，一个是开关作用。在这里,我们只了解它跟本电路有关的开关作用。首先把三极管想成一个水龙头，上面的 Vcc 就是水池，继电器是一个水轮机，下面的 GND 是比水池低的任何一点。刚才说过，三极管就是水龙头，它的把手就是那个带有电阻的引脚。现在，单片机的某一个需要控制这个继电器电路的输出引脚就是一只"手"，当单片机的这个引脚输出低电平的时候，就像"手"在打开三极管"水龙头"，水就从上往下流,继电器"水轮机"就开始转起来了。反之，如果是输出高电平，"手"就开始关"水龙头"，继电器"水轮机"因为没有水流下来，就会停止，这就是三极管的开关作用。简单的理解和记忆就是：三极管是一个开关器件，其实你真的可以将它看成是一个开关，只不过它不是用手来控制，而是用电压(电流)来控制的，因此三极管有些时候也被称做电子开关(与机械开关相区别)。

D1 是保护二极管，如果不需要深入理解的话，你大可不必追就为什么有它存在，但是一定得记住，只要是用三极管驱动继电器的场合，一般都有它的存在.需要特别注意的是它的接法：并联在继电器两端，阴极一定是接 Vcc。

3.27.4 例程 01 继电器 1 秒种切换一次

代码如下：

```
/*******************************************************
* 例程：控制继电器开关
* 作者：www.armjishu.com
* 版本：v1.0
```

```
* 内容：对应的继电器接口需用杜邦线连接
*******************************************************/
#include<reg52.h> //包含头文件，一般情况不需要改动，头文件包含特殊功能寄存器的定义
sbit RELAY1 = P1^1;//定义继电器信号输出端口 1
sbit RELAY2 = P1^2;//定义继电器信号输出端口 2
/*------------------------------------------------
 mS 延时函数，含有输入参数 unsigned char t，无返回值
 unsigned char 是定义无符号字符变量，其值的范围是 0~255
 这里使用晶振 12M，精确延时请使用汇编
------------------------------------------------*/
void DelayMs(unsigned char t)
{
    while(t--)
    {
        //大致延时 1mS
        DelayUs2x(245);
        DelayUs2x(245);
    }
}
/*-- 主函数 --*/
void main (void)
{
    while(1)
    {/* 两个继电器不停的切换操作 */
        RELAY1=!RELAY1;
        RELAY2=!RELAY2;
        DelayMs(1000);
    }
}
```

硬件连接关系如下表 3-120 所示：

表 3-120 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P1.1 和 P1.2	JP13	直连	JP9 第 4 脚和 JP9 的第 5 脚	杜邦线连	控制继电器
实验现象：下载程序后，可以听到两个继电器不停的咔咔的响声，这个就是表示继电器在不停的进行切换操作					

知识要点：

1. 当 K_IN2 管脚输出高电平的时候，三极管导通，通过 VCC 输出电流给继电器驱动继电器开关吸合；当 K_IN2 管脚输出低电平的时候，三极管截止，继电器恢复，如下图 3-180 所示：

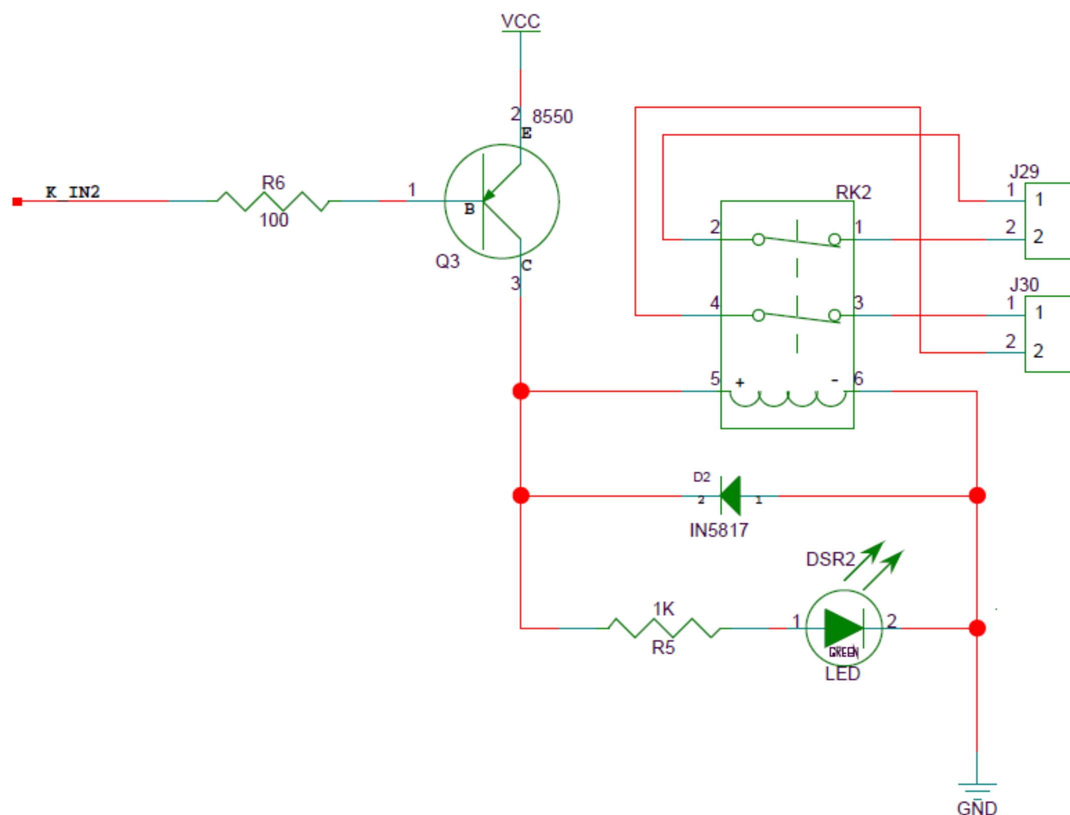


图 3-180 继电器原理图

2.K_IN2 连着位号为 Q3 的 8550 三极管的基极，三极管 E 发射极连 VCC，集电极 C 连继电器，当 K_IN2 使用一个很小的电流，就可以将三极管 Q3 打开，使得 E 发射极和集电极 C 之间导通，电流驱动能力被放大，达到足够驱动继电器的电流电压值，这里如果还有不清楚的可以去阅读一下三极管的相关知识。

3.在 while()循环里，每延时一段时间，控制继电器的管脚电平就会输出相反的电平，所以继电器就被不断的关闭和打开，程序运行后可以听到继电器发出“咔咔咔”的响声，每发出一次表示继电器产生一次切换操作。

3.27.5 更多有关继电器的例程

更多有关继电器相应的例程可通过北京航空航天大学出版社下载专区下载，如下表 3-121：

表 3-121 继电器更多丰富例程介绍（含详细注释和文档分析）

序号	例程功能
例程 01	继电器 1 秒种切换一次
例程 02	任意按键控制继电器开关
例程 03	一个按键控制一路继电器开关

3.28 315M无线模块

3.28.1 无线通信

无线通信主要包括微波通信和卫星通信。这里主要是讲解微波通信，微波是一种无线电波，通常又被称为电磁波，它传送的距离一般只有几十千米。但微波的频带很宽，通信容量很大。微波通信每隔几十千米要建一个微波中继站。卫星通信是利用通信卫星作为中继站在地面上两个或多个地球站之间或移动体之间建立微波通信联系。

所以说无线通信是利用电磁波信号可以在自由空间中传播的特性进行信息交换的一种通信方式。

采用红外技术的产品有方向性，发射器必须对准接收器，并且中间不能有阻挡物。距离一般不超过 7 米、不受电磁干扰,红外类产品的优势是产品成本低、价格便宜；而采用无线通信技术的产品是用无线电波来传送控制信号的，它的特点没有方向性，可以不“面对面”控制、距离远，可达数十米。发射器和接收器之间只要没有能起屏蔽作用的金属阻挡物，就可正常使用。射频技术的产品成本通常要高一些，但其无方向性，使用更方便，所以更受用户的欢迎。

3.28.2 无线模块简介

无线模块是利用无线技术进行无线传输的一种模块。它被广泛地应用于电脑无线网络，无线通讯，无线控制等领域。无线模块主要由发射器，接收器和控制器组成。

在无线通信技术出来之前，很多的操作都需要用到人手去操作，如开关门，特别是一些比较大的门，开关起来不方便，控制设备机器工作的时候也需要人手去控制，可能会有危险的情况发生，还有一些设备之间的相互通信也是用到信号线的连接进行通信，受到距离、准确性的一些因素的影响。为了解决这些问题，无线通信技术出现了。

无线数据传输广泛地运用在车辆监控、遥控、遥测、小型无线网络、无线抄表、门禁系统、小区传呼、工业数据采集系统、无线标签、身份识别、非接触 RF 智能卡、小型无线数据终端、安全防火系统、无线遥控系统、生物信号采集、水文气象监控、机器人控制、无线 232 数据通信、无线 485/422 数据通信、数字音频、数字图像传输等领域中。

无线模块主要特点如下：

1.供电电压。电压越高功耗越大，相应传输距离可能越远；电压越低，功耗越低，当然距离可能就相对较近；是否带天线都影响其模块的功耗大小。

2.模块地址。模块是否有地址，有些简单的无线模块是通过跳帽拉高拉低来设置地址；有些稍微智能一点的模块是芯片内部设置模块地址；

3.传输编码。传输的编码模式各个模块和芯片厂家设定都不同，编码是一种收和发彼此约定的通信方式，编码的好坏可以灵活纠正传输中的错误，编码包括纠错码，包括地址放在什么地方，内容放到哪里等。

4.接收灵敏度。接受灵敏度描述了无线模块接收微弱信号的能力，不同的模块以及不同的无线芯片其接收灵敏度各不相同。

5.发射功率和功耗。发射功率越大，信号强度越强，同时发射距离也相对较远，但在足够的范围内，发射功率越低对模块来说功耗就越低。所以如果范围被确定，尽量选择功率较低的模块会更好。

6.频段范围。无线芯片一般都有个频率范围，例如某 433M 无线模块上的芯片的频率范围是 119-1050MHZ，可以定制，根据芯片手册来进行设置其他的频段。

7.对外接口。模块对外可以是 SPI，IIC 或者串口来与之通信，其他设备通过这个接口与无线模块通信，使得这些其他设备具备无线通信的能力。

3.28.4 无线接收原理

无线通信是靠无线微波或者又叫无线电波来传输信息，无线电波是一种能量传输形式，在传播过程中，电场和磁场在空间是相互垂直的，同时这两者又都垂直于传播方向。

无线通信速度。无线电波和光波一样，它的传播速度和传播媒质有关，无线电波在真空中的传播速度等于光速。

无线电波波长。无线电波的速度除以无线电波的频率就是无线电波的波长；速度是 1 秒钟传输多少米，频率是 1 秒钟多少赫兹，波长的单位也是米。

天线。天线把需要传输出去的电信号转化成无线电波发射到空间，另外一端的接收天线收到无线电波后将其转化成电信号。

障碍物。电波在传播途径上遇到障碍物时，总是力图绕过障碍物，再向前传播。手机信号属于超短波或微波，波长较短，在地面衰减很快；而类似 315M 通信的无线电波波长较长，穿透力要强一点，传输较远。也就是说，频率越高、建筑物越高、越近、影响就越大；相反，频率越低、建筑物越矮、越远、影响就越小。

电流通过导线形成电磁波辐射，辐射能力与导线长短以及形状有关，当导线长度增大到可与波长相比拟时，导线上的电流就大大增加，因而就能形成较强的辐射。无线电波的产生原理可以参看图 3-181 所示：

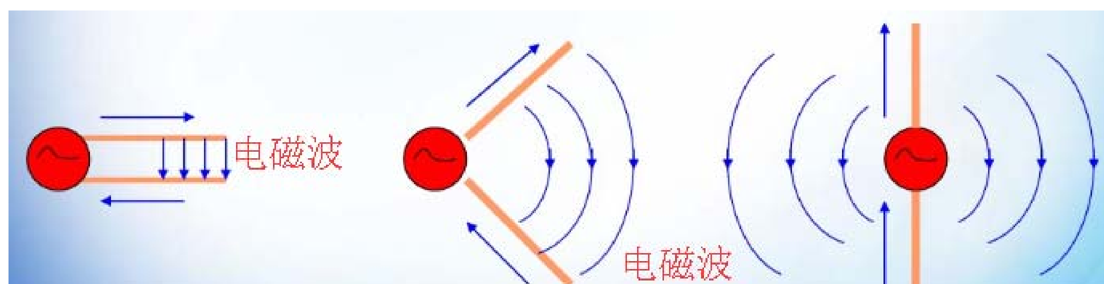


图 3-181 无线辐射电磁波原理图

可以看到当天线越闭合，发出的电磁波的波长越短；相反的，如果天线拉到一条直线，波长是最长的。波长的变化就会导致频率的变化，比如 315M 通信，315M 就是通信的频率，这样只需要调整对应天线，配置为与 315M 频段吻合的波长就可以彼此接收和发射信号了。

3.28.5 315M无线模块

无线模块在市场上分为两种，一种是仅仅实现了无线收发通信的模块，而另外一种在无线收发基础上还增加编码解码功能，采用一种通信编码算法更加保证数据传输的稳定性。这两种模块的区别就是，仅仅实现了无线收发通信的模块在传输数据时候需要自己编码和校

验，而带了编码和解码功能的模块只需要去控制编码解码芯片，芯片就会发送一系列对应的命令到发射信号区域将该命令发射给接收模块，接收模块接收到一系列的信号转化成电信号，再给自己的解码芯片进行解码，从而获得发射和接收模块的地址以及内容。

315M 无线模块的工作电压范围一般是 3～12V，当电压变化时发射频率基本不变，和发射模块配套的接收模块无需任何调整就能稳定地接收。当发射电压为 3V 时，空旷地传输距离约 20～50 米；当电压 5V 时约 100～200 米，当电压 9V 时约 300～500 米，当发射电压为 12V 时，为最佳工作电压，具有较好的发射效果，发射电流约 60 毫安，空旷地传输距离 700～800 米，发射功率约 500 毫瓦。当电压大于 l2V 时功耗增大，有效发射功率不再明显提高。这套模块的特点是发射功率比较大，传输距离比较远，比较适合恶劣条件下进行通讯。

315M 频率传输电磁波的波长是多少呢？现在先计算一下波长，波长=光速/频率=300/315=0.952 米，那么 1/4 波长需要的天线长度=波长*1/4=0.952/4=0.238 米，考虑导线传播高频信号的缩短率在 0.98 左右，因此天线长度=0.238*0.98=0.23，所以经过计算后一般的 315M 天线长度选择 23cm，433M 的天线长度在 17cm 左右，最短必须这么长，至少要够四分之一波长。

315M 发射模块经过天线发射出对应波长的电磁波，被另外一个 315M 接收模块接收到，因为收发模块都是使用能发出一样电磁波波长的天线，所以彼此就能发射和接收到电磁波信号，接收模块接收到电磁波信号之后，通常情况下，可能接收的电磁波比较弱，还需要在接收到之后对接收到的电磁波进行处理，比如需要放大，修整；不排除在传送过程有存在干扰现象，个别数据出错，这时候还需要对电磁波转化完之后还要对数据进行校验，所以这就涉及了编码规则了，一套好的机制可以使得数据的传输正确率大大提高。

3.28.6 带编码解码的 315M无线模块

编码是将源对象内容按照一种标准转换为一种标准格式内容,解码是和编码对应的，它使用和编码相同的标准将编码内容还原为最初的对象内容；有了带编码解码的芯片，就只需要设置和控制这个芯片，使得芯片去控制发射天线发射一套事先确认好的指令编码就可以了，这样接收方也同样接收到这套指令编码，再通过解码解出数据内容来；这样的好处是降低了研发难度和工作量，就好比是自己做菜，需要从土豆切丝开始慢慢切菜，而现在有了一个酒店只需要一个电话就可以订好一桌子菜一样，更加省力了，这些编码事先都被芯片厂家封装到了芯片内部，通过查阅芯片手册就可以控制芯片的编码。接下来详细分析一下带编码解码的 315M 模块，可以看到下图 3-182：

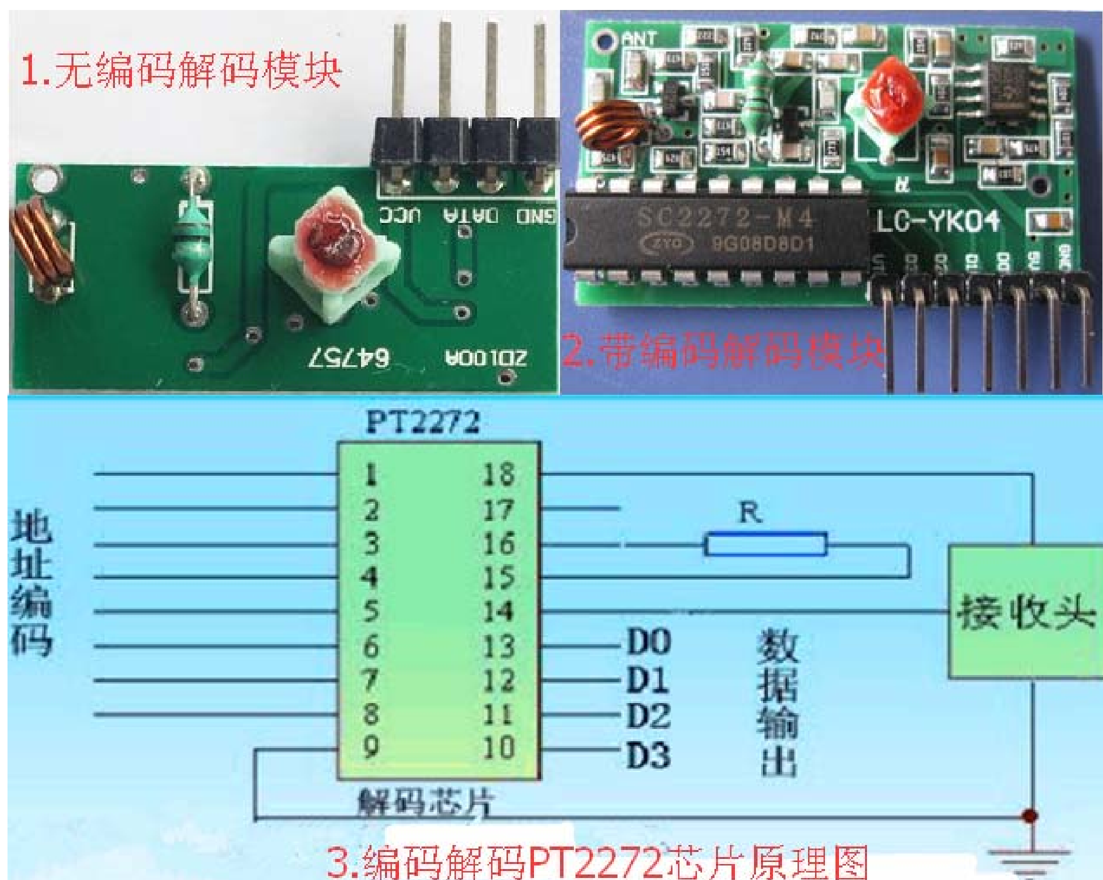


图 3-182 315M 无线模块外观和编码解码原理图

里面有描述了三张图，1 是无编码解码模块，模块上没有编码解码芯片；2 是带编码解码的模块，可以看到模块上有个 2272 型号的长方体芯片；3 是 2272 芯片的使用常规原理图。

分析 2272 芯片，芯片上分别有 D0、D1、D2、D3 为信号管脚可以做为输出使用，当某个按键按下后，相应的数据端口就输出高电平，在这几个端口加一级放大就可以驱动继电器，功率三极管，进行负载遥控开关控制；另外也可以直接连到单片机的 I/O 脚上，通过单片机采集数据端口状态，然后进行外部控制。315M 无线模块由发射部分和接收部分组成：

关于无线通信的地址设置，在通常使用中，一般采用 8 位地址码和 4 位数据码，这时解码芯片 PT2272 的第 1～8 脚为地址设定脚，有三种状态可供选择：悬空、接正电源、接地三种状态，地址编码不重复度为 3 的 8 次方是 6561 组，只有发射端 PT2262 和接收端 PT2272 的地址编码完全相同，才能配对使用，遥控模块的生产厂家为了便于生产管理，出厂时遥控模块的 PT2262 和 PT2272 的八位地址编码端全部悬空，这样用户可以很方便选择各种编码状态，用户如果想改变地址编码，只要将 PT2262 和 PT2272 的 1～8 脚设置相同即可，例如将发射机的 PT2262 的第 2 脚接地，第 3 脚接正源，其它引脚悬空，那么接收机的 PT2272 只要也第 2 脚接地，第 3 脚接正电源，其它引脚悬空就能实现配对接收。地址设置跳线如图 7 所示，用户可以在 PCB 板上直接将地址引脚（PCB 板中间 8 个过孔焊盘）与 L（低电平）或 H（高电平）相连，从而实现地址设置。PT2262 与 PT2272 地址设置要完全一样。当两者地址编码完全一致时，接收机对应的 D0～D3 端输出约 4V 互锁高电平控制信号，同时 VT 端也输出解码有效高电平信号。

3.28.7 例程 01 315M无线模块任意按键控制LED实验

程序如下：

```
/*********************************************************************
* 例程：315M 无线任意按键控制 LED
* 作者：www.armjishu.com
* 版本：v1.0
* 内容：通过 315M 遥控器的按键控制神舟 51 开发板上的 LED 状态
* 现象：本实验是通过 51 单片机的 IO 管脚连接 315M 无线接收模块，接收遥控器的
*       按键信息。通过 315M 遥控器的任意按键按下时，接收模块的 VT 信号有效
*       此时点亮神舟 51 开发板上的所有 LED，否则熄灭所有 LED。
*********************************************************************/
/* 包含头文件 */
#include <reg52.h>
#define   WL_315M_PORT    P0     //315M 无线接收模块对应端口
sbit WL_315M_VT = P0^7;          //定义 VT 接收数据有效端口
#define   LED_PORT        P2     //定义 led 输出端口
/*-- 主函数 --*/
void main (void)
{
    WL_315M_PORT = 0xff;       // 315M 无线接收模块对应端口电平置高
    LED_PORT = 0xff;           // 熄灭全部 led 指示灯
    while (1)                  // 主循环
    {
        if(WL_315M_VT)         // 如果检测到高电平，说明遥控器有按键按下
        {
            LED_PORT = 0x00;   // 如果按键按下 led 点亮
        }
        else
        {
            LED_PORT = 0xFF;   // 如果按键没有按下 led 熄灭
        }
    }
}
```

神舟51开发板的配置如下表3-122所示。

表 3-122 硬件连接关系

用排线电缆或杜邦线连接“单片机 IO”和“模块接口”					
单片机接口	插座 1	方式	插座 2	线缆	功能
P0	JP15(A 向左)	直连	JP12(A 向上)	8 芯排线	315M 无线模块信号

P2	JP16(A 向左)	交叉	JP19(A 向左)	8 芯排线	2.4G 无线模块 2 信号
实验现象：实验前先对开发板断电，安装好 315M 无线模块。上电后如果 315M 无线遥控的任意按键按下时，接收模块的 VT 信号有效，此时点亮神舟 51 开发板上的所有 LED，否则熄灭所有 LED，则说明实验成功，否则实验失败。					

连接图如下图3-183所示：

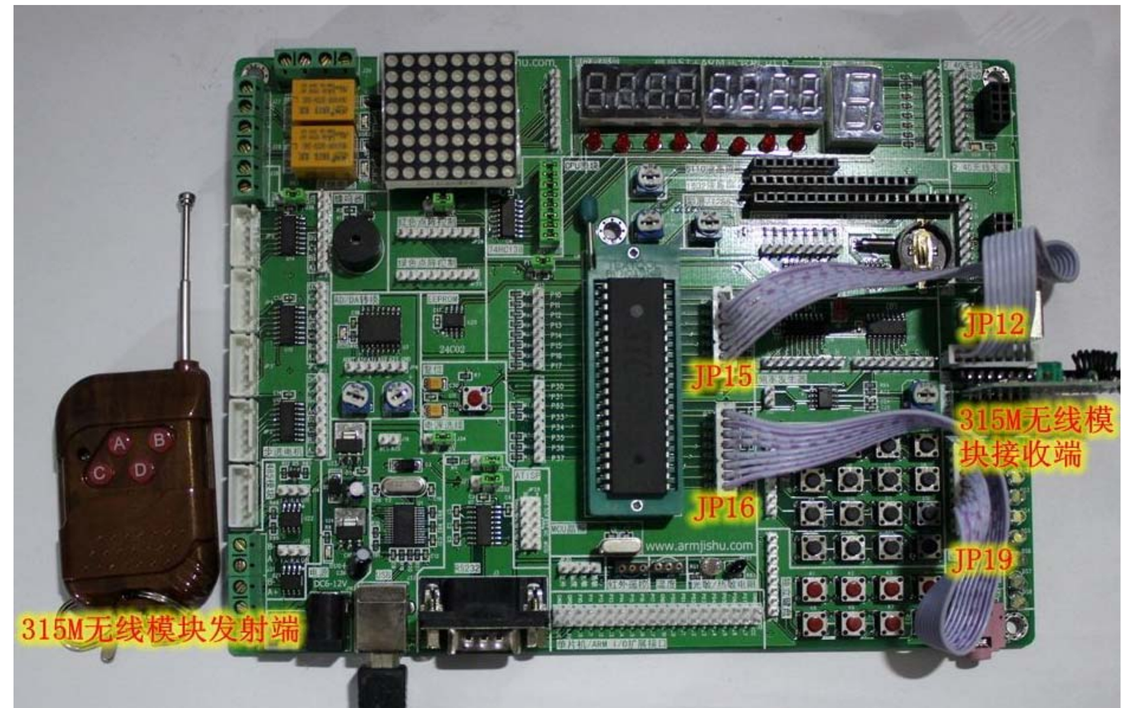


图 3-183 硬件连接实物图

知识要点：

本实验通过 315M 遥控器的按键控制神舟 51 开发板上的 LED 状态。通过 51 单片机的 IO 管脚连接 315M 无线接收模块，接收遥控器的按键信息。通过 315M 遥控器的任意按键按下时，接收模块的 VT 信号有效，此时点亮神舟 51 开发板上的所有 LED，否则熄灭所有 LED。

一般来说 315M 无线接收模块不需要对按键去抖，但是必须首先检测 VT 信号有效后再判断按键 D0-D3 信号状态。

说明：315M无线接收模块的D0-D3与315M遥控器的ABCD四个按键是一一对应的关系，但是具体哪个按键对应哪个管脚视模块而定。

3.28.8 更多有关 315M无线模块的例程

更多 315M 无线模块相应的例程可通过北京航空航天大学出版社下载专区下载，如下表 3-123：

表 3-123 315M 无线模块更多丰富例程介绍（含详细注释和文档分析）

序号	例程功能
例程 01	315M 无线模块任意按键控制 LED 实验
例程 02	315M 无线任意按键控制 LED 状态转换
例程 03	315M 无线 4 个按键控制 LED

例程 04	315M 无线 4 个按键控制 LED 状态转换
例程 05	315M 无线按键加减操作数码管显示

3.29 2.4 G无线模块

3.29.1 低速和高速无线模块区分

低速模块一般最高传输速率为 150kbit/s 以下，它们的工作频段一般在 315/433/915MHz，这个频段大家是不是很熟悉呀？常用的 PT2262 与 PT2272 无线遥控器使用的就是 315MHz 或者 433MHz 频段，在这个频段传输数据有一个优点，就是穿透能力强，距离可以相对远一些，能传输数百米。但是由于应用范围广，满大街的汽车电子锁都用这个频段，干扰能不大吗？因此数据传输的速率也不能太高。

高速模块一般是指传输速率在 500kbit/s 以上的无线模块，一般工作在 2.4GHz 频段，这个频段干扰相对比较少，但是由于频率越高，穿透能力越弱，2.4G 无线模块一般工作在直线无遮挡的环境下，距离大致在几十米。

两种模块各有优缺点，互为补充，本章专门研究 2.4GHz 频段的无线高速数传模块。相对于低速模块，它们的普及程度离我们更近，在不久的将来应用会越来越广泛，例如现在的智能云，用手机控制电饭煲，洗衣机，智能家居都是使用这类频段的无线模块，WIFI 模块就是使用 2.4G 频段。

3.29.2 2.4G无线通信智能门锁产品原理

进一步了解 2.4G 无线通信的作用，这里举一个小例子；小明的新家希望安装一款智能门锁，到市场了解，主要有三种类型：

1.用 315M 连接的智能门锁，会配置一个遥控器，在门口可以通过遥控器或者钥匙都可以让门锁打开或关闭；价格便宜，通信简单，只有开或者关两个操作。

2.用 2.4G 无线通信实现的智能门锁还配带门铃，支持语音对讲功能，并且房间内的语音接收机是一台类似手机的设备，可以手持对话以及手持该设备进行开关门；因为 2.4G 频段比较高，可以传输比 315M 的更多的数据，所以适合语音通信或者视频通信；另外这种通信是一对一的，而且带有加密功能，比较安全。

3.基于 WIFI 的智能门锁和门铃，目前大部分 WIFI 也属于 2.4G 频段范畴，它与常规其他 2.4G 通信不同的是 WIFI 可以组网，也就接入 internet，支持云端，也就是说你人在北京可以使用手机把在上海家里的门口打开；好处当然是更加方便灵活，弊端显而易见的就是安全性比较难以掌握。

小明考虑到没有出差的习惯，生活比较简单，没有必要选择 WIFI 的智能门锁，最后选择了第二种可以支持语音通信的智能门锁，小明选择的这款产品就是本节讨论的 2.4G 技术，学懂了这个再去了解 WIFI 无线模块就会比较容易，WIFI 无线模块比本节讨论的 2.4G 无线模块多了一个 TCP/IP 协议栈，因为 WIFI 无线模块就像一台上网的设备一样，需要接入到互联网中，本章讨论的内容不包括 TCP/IP，只是简单的 2.4G 频段的无线通信。

3.29.3 2.4G无线通信模块的特性

频段。2.4G无线模块（英文：2.4Ghz RF transceiver / receiver module）工作在全球免申请ISM频道2400M-2483M范围内，实现开机自动扫频功能，共有50个工作信道，可以同时供50个用户在同一场合同时工作，无需使用者人工协调、配置信道；同时，高效GFSK调制，抗干扰能力强，特别适合工业控制场合，126频道，满足多点通信和跳频通信需要，内置硬件CRC 检错和点对多点通信地址控制。

距离。实际使用经验，建议距离在 100 米范围左右比较合适。

速率。最高工作速率 2Mbps，最低工作速率 250K ，

功耗和供电。低电压供电：1.9V~3.6V；nRF24L01 内置频率合成器、功率放大器、晶体振荡器、调制器等功能模块。其功耗低，在以-6dBm 功率发射时，工作电流只有 9mA 左右；而接收时，工作电流只有 12mA 左右

模块地址。模块可软件设地址，只有收到本机地址时才会输出数据。

3.29.4 2.4G无线模块分析

由下图可以看到，nRF24L01无线模块引出了8个管脚，具体下图所示：

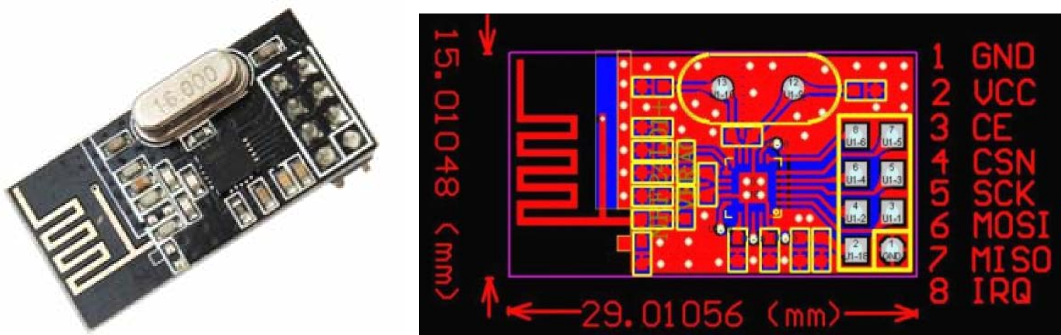


图 3-184 2.4G 无线模块

下表 3-124 为 2.4G 无线模块 nRF24L01 管脚说明：

表 3-124　2.4G 无线模块管脚说明

管脚信号	模块管脚	信号说明
GND	1	地信号
VCC	2	电源输入　电压范围1.9V~3.6V电源
CE	3	高电平有效，（RX）发射或(TX)接收模式控制
IRQ	8	低电平有效，中断输出。
CSN	4	SPI片选信号　（SPI信号）
SCK	5	SPI时钟，由主器件产生　（SPI信号）
MOSI	6	从SPI数据输出，三态输出。　（SPI信号）

MISO	7	从SPI数据输入　　　　　　　　（SPI信号）

可以看到，2.4G模块可以使用SPI协议，模块提供排针可以直接插入到控制板上进行通信，最有特色的是那个奇怪的天线。

接下来分析一下天线的情况，天线分为PCB天线和板外天线，之前315M无线通信章节使用的是外接天线，外接天线有很多种类，这里暂不讨论，这里只分析PCB天线，即在PCB上画出铜丝的走线作为天线使用。

两款2.4G无线PCB天线原理图如下图3-185所示：

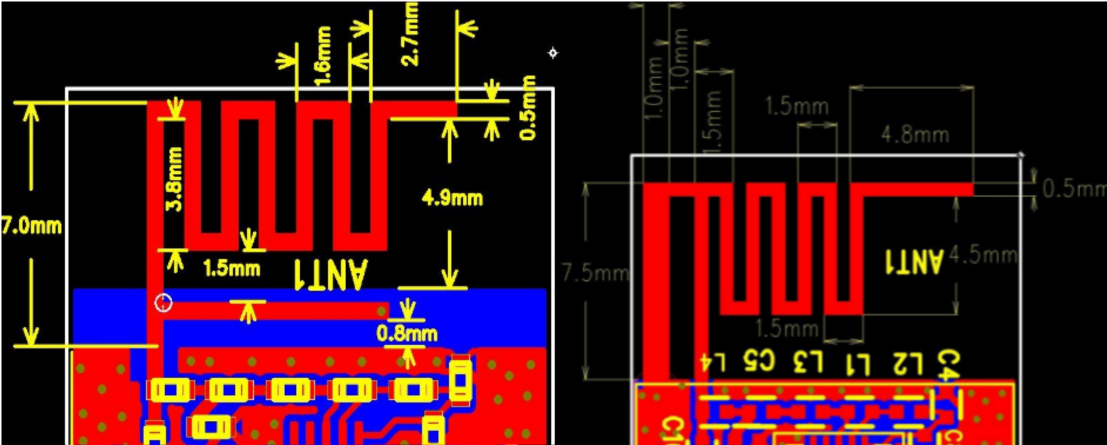


图3-185 两款2.4G无线PCB天线原理图

通过对比可以知道，左边这款就是本章节讨论的2.4G无线模块的PCB天线走法，右边是另外一种走法。

左边名称叫wiggle天线。整个天线的高度是7.0mm，具体尺寸都在里面有详细的标注，板材要求是两层FR4，板厚度是1.0mm的，板子厚度和大小都可能影响到无线的性能，最后端增加了2.7mm的长度供调试天线用；这个PCB天线所占PCB空间比较小，增益也稍差一点，可以用于对体积要求比较小的无线终端，比如对空间要求比较紧凑的无线鼠标设备。

右边叫PIFA天线。也是类似，后端增加了4.8mm长度供调试天线用；这个PCB天线所占PCB空间很大，信号增大增益也是最强的，如果PCB面积足够，建议使用这种天线。

如果需要设计无线模块的PCB天线，切记一定要按照规范来操作，这些都是芯片官方厂家得出的经验值，没有理由，如果要设计就按照上面的标准尺寸画出PCB即可，当然不同的PCB工艺对其是有一定影响的，到时候再在研发过程中进行调整。

3.29.5 nRF24L01 芯片工作原理

为了分析 2.4G 无线芯片的功能，先分解一下芯片所要做的工作；假设一个 2.4G 无线模块需要接受一组数据，那么接收数据的流程基本上是从无线模块的天线接收到模拟信号，模拟信号从天线管脚传入到 2.4G 芯片内部，在内部再转化成数字信号，数据可能是有规则的编码，那么就需要按规则来进行解码解出实际数据内容，再经过校验和纠偏，再把对应的数据存入到芯片内部的缓冲中，芯片是 SPI 接口的，则通过缓冲中的数据通过 SPI 的方式传出芯片到外部的单片机中，这样就完成了整个数据接收过程。

下面分析一下实际是如何处理的，如nRF24L01内部结构如下图3-186所示：

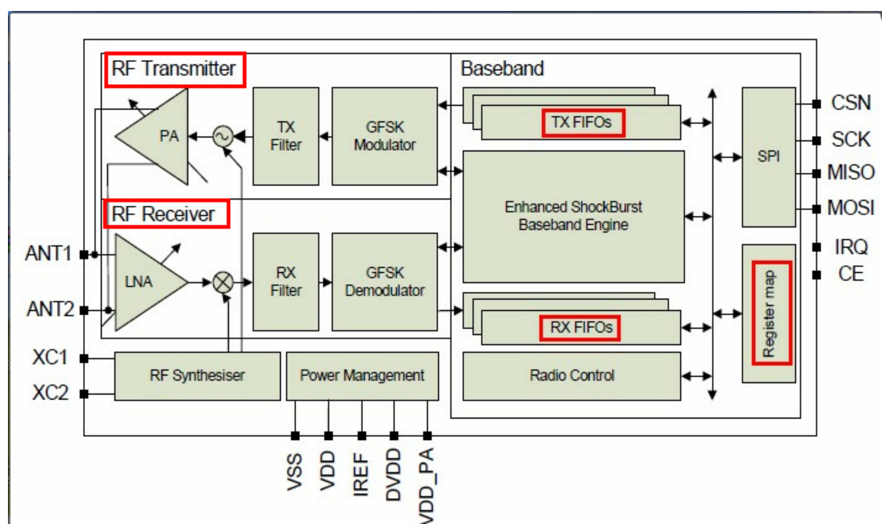


图 3-186 2.4G 无线模块内部结构

具体接收流程：模拟数据被天线 ANT 接收进入 LNA 转化成数字信号，该数字信号进入到 RX Filter 过滤器和 GFSK 解码器，解码之后出来了真正的数据放入到 RX FIFOs 中提供给 SPI 接口来读取。发射流程也是大同小异。

TX FIPOs、RX FIFOs 分别用于存储待发送和接收到的数据包。RF Transmitter 为射频发射器，RF Receiver 为射频接收器。XC1、XC2 连接晶体振荡器，用于对信号的模数/数模转换等提供时钟。ANT1、ANT2 天线接口。

51 单片机通过 SPI 接口（这里用 I/O 口模拟）和 nRF24L01 无线模块通信，Register map 是用来保存单片机对 nRF24L01 无线模块配置的寄存器，通过这些寄存器来达到控制无线模块工作。

3.29.6 nRF24L01 无线模块工作模式

工作模式决定了模块和模块间的通信方式，主要由 51 单片机对 nRF24L01 无线模块的配置决定。下图为 nRF24L01 无线模块的工作模式。它们由 CE 和寄存器内部 PWR_UP、PRIM_RX 共同控制，有发射模式、接收模式、空闲模式、掉电模式。如下图表 3-125：

表 3-125 2.4G 无线模块工作模式

模式	PWR_UP	PRIM_RX	CE	FIFO 寄存器状态
接收模式	1	1	1	-
发射模式 I	1	0	1	数据在 TX FIFO 寄存器中
发射模式 II	1	0	1→0	停留在发射模式，直至数据发送完
待机模式 I	1	-	0	无正在传输的数据
待机模式 II	1	0	1	TX FIFO 为空
掉电模式	0	-	-	-

空闲模式。nRF24L01 的空闲模式是为了减少平均工作电流而设计的，其最大的优点是，实现节能的同时，缩短芯片的启动时间。在空闲模式下，部分片内晶振仍在工作，此时的工作电流跟外部晶振的频率有关。

待机模式。待机模式 I 已经进入休眠，没有正在传输的数据；而待机模式 II 则必须要求 TX FIFO 寄存器为空才可以，待机模式下，所有配置字仍然保留，在该模式下晶体振荡器仍然是工作的。

掉电模式。在掉电模式下，配置字的内容仍保存在 nRF24L01 片内，这是它和断电状态的最大区别，此时工作电流为 900mA 左右。

接收和发射模式。这些模式规定了无线模块间进行通信的相关协议，这些协议的规定的目的是保障通信的正确率，降低数据出错概率，纠正纠偏，在保证数据正确性的前提下尽量提高数据传输速度而设计的。这里只把设计的内容写出来，大家可以通过这些规定的流程方法可以自行分析一下这样规定数据传输协议有什么好处，对数据传输的速度和正确性有什么帮助。

通过 51 单片机对 nRF24L01 无线模块的配置，收发模式可以分为三种，分别为：Enhanced ShockBurstTM 收发模式、ShockBurstTM 收发模式和直接收发模式三种。工作在直接收发模式下时，数据一般在低频状态下进行设置，以保证接收机能探测到信号。Enhanced ShockBurstTM 收发模式和 ShockBurstTM 收发模式，前者比后者多了一个确认数据传输的信号，保证数据传输的可靠性。如下表 3-126 为两种模式的数据包格式：

表 3-126 两种模式的数据包格式

增强 ShockBurst 数据包				
开始	地址 3~5 字节	标志位 9 位	有效载荷 1~32 字节	CRC 0 / 1 / 2 字节
ShockBurst 数据包				
开始		地址 3~5 字节	有效载荷 1~32 字节	CRC 0 / 1 / 2 字节

下面详细分析 Enhanced ShockBurstTM 收发模式，通过对这个模式的分析了解，模块间进行数据交换的流程。

Enhanced ShockBurstTM 收发模式下，使用片内的先入先出堆栈区，数据低速从微控制器送入，但高速(1Mbps)发射，这样可以尽量节能，因此，使用低速的微控制器也能得到很高的射频数据发射速率。与射频协议相关的所有高速信号处理都在片内进行，这种做法有三大好处：尽量节能；低的系统费用(低速微处理器也能进行高速射频发射)；数据在空中停留时间短，抗干扰性高。Enhanced ShockBurstTM 技术同时也减小了整个系统的平均工作电流。

在 Enhanced ShockBurstTM 收发模式下， NRF24L01 自动处理字头和 CRC 校验码。在接收数据时，自动把字头和 CRC 校验码移去。在发送数据时，自动加上字头和 CRC 校验码，在发送模式下，置 CE 为高，至少 10us，将时发送过程完成后。

Enhanced ShockBurstTM 发射流程

A. 把接收机的地址和要发送的数据按时序送入 NRF24L01；

B. 配置 CONFIG 寄存器，使之进入发送模式。

C. 微控制器把 CE 置高（至少 10us），激发 NRF24L01 进行 Enhanced ShockBurstTM 发射；

D.N24L01 的 Enhanced ShockBurstTM 发射

(1) 给射频前端供电；

(2)射频数据打包(加字头、CRC 校验码)；

(3) 高速发射数据包；

(4)发射完成，NRF24L01 进入空闲状态。

Enhanced ShockBurstTM 接收流程

A. 配置本机地址和要接收的数据包大小；

B. 配置 CONFIG 寄存器，使之进入接收模式，把 CE 置高。

C. 130us 后，NRF24L01 进入监视状态，等待数据包的到来；

D.当接收到正确的数据包(正确的地址和 CRC 校验码)，NRF2401 自动把字头、地址和 CRC 校验位移去；

E. NRF24L01 通过把 STATUS 寄存器的 RX_DR 置位(STATUS 一般引起微控制器中断)通知微控制器；

F. 微控制器把数据从 NewMsg_RF2401 读出；

G. 所有数据读取完毕后，可以清除 STATUS 寄存器。NRF2401 可以进入四种主要的模式之一。

nRF24L01 在掉电模式下转入发射模式或接受模式前必须经过 1.5mS 的待机模式。断电后寄存器内容丢失，模块上电需重新配置。具体配置时序如下表 3-127：

表 3-127　2.4G 无线模块的时序配置

Nrf241L01 时序	最大值	最小值	参数名
掉电模式——待机模式	1.5ms		Tpa2stdy
待机模式——发送/接收模式	130us		Tstdy2a
CE 高电平保持时间		10us	Thee
CSN 为低电平，CE 上升沿的延迟时间		4us	Tpece2csn

3.29.7 单片机SPI访问 2.4G无线模块

单片机通过 SPI 通信接口来访问 2.4G 无线模块；SPI 口为同步串行通信接口，有专门的数据管脚，可以直接对 2.4G 无线模块发送控制指令，这些控制指令由 nRF24L01 的 MOSI 输入，相应的状态和数据信息是从 MISO 输出给 MCU；控制指令如下表 3-128：

表 3-128　2.4G 无线模块的控制指令

SPI 接 口 指 令		
指令名称	指令格式	操作
R_REGISTER	000A AAAA	读配置寄存器。AAAAA 指出读操作的寄存器地址
W_REGISTER	001A AAAA	写配置寄存器。AAAAA 指出写操作的寄存器地址只能在掉电模式或待机模式下操作
R_RX_PAYLOAD	0110 0001	读 RX 有效数据：1-32 字节。读操作全部从字节 0 开始。当读 RX 有效数据完成后，FIFO 寄存器中有效数据被清除，应用于接收模式下。
W_RX_PAYLOAD	1010 0000	写 TX 有效数据：1-32 字节。写操作从字节 0 开始。应用于接收模式下
FLUSH_TX	1110 0001	清除 TX FIFO 寄存器，应用于发射模式下。
FLUSH_RX	1110 0010	清除 RXFIFO 寄存器，应用于接收模式下。在传输应答信号过程中不应该执行此指令，也就是说，若传输应答信号过程中执行此指令的话将使得应答信号不能被完整的传输。
REUSE_TX_PL	1110 0011	应用于发射端。 重新适应上一包发射的有效数据。当 CE=1 时，数据被不断重新发射。

		在发射数据包过程中必须禁止数据包重利用功能。
NOP	1111 1111	空操作。可用来读状态寄存器。

nRF24L01 的收发模式由配置字决定。nRF24L01 所有的配置字都由配置寄存器定义，这些配置寄存器可通过 SPI 口访问。nRF24L01 寄存器有 25 个，常用的配置寄存器如下表 3-129。

表 3-129　**nRF24L01 寄存器**

地址（H）	寄存器名称	功能
00	CONFIG	设置 24L01 工作模式
01	EN_AA	设置接收通道及自动应答
02	EN_RXADDR	使能接收通道地址
03	SETUP_AW	设置地址宽度
04	SETUP_RETR	设置自动重发数据时间和次数
07	STATUS	状态寄存器，用来判定工作状态
0A~0F	RX_ADDR_P0~P5	设置接收通道地址
10	TX_ADDR	设置发送地址（先写低字节）
11~16	RX_PW_P0~P5	设置接收通道的有效数据宽度

下面分析一下 51 单片机和 nRF24L01 无线模块通信所遵循的 SPI 时序，当 CSN 引脚由低电平向高电平跳变时结束当前 SPI 的活动；当 CSN 引脚由高电平向低电平跳变时；SPI 接口开始等待一条指令。

读时序。其中数据 Cn 表示 SPI 指令位；数据 Sn 表示状态寄存器位；数据 Dn 表示数据位。每个字节高位在前；SPI 读时序和写时序如下图 3-187 和图 3-188 所示：

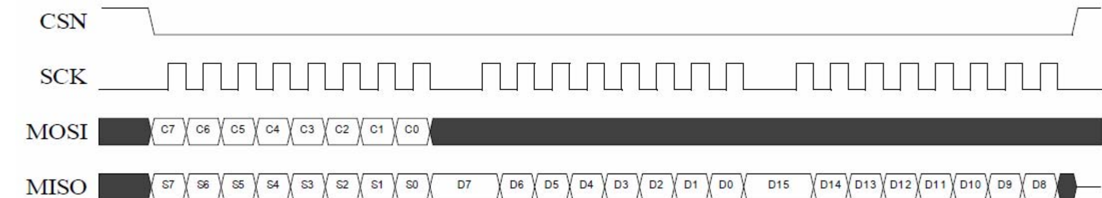


图 3-187　**SPI 读时序**

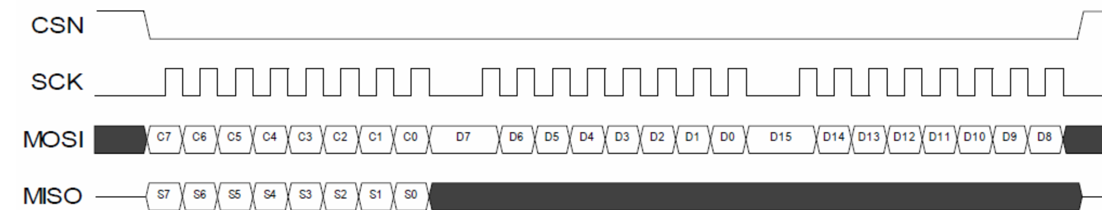


图 3-188　SPI 写时序

SCK 时钟线，由低电平向高电平跳变，高电平时取数据，无论是读操作还是写操作，MISO 线都返回状态寄存器位 Sn。

发送数据时，首先将 nRF24L01 配置为发送模式；接着把发送地址和发送数据按照时序要求经过 SPI 总线写入 nRF24L01 缓存区，发送数据必须在 SPI 片选 CSN 为低时，连续写入。而发送地址在发射时写入一次即可，然后使能管脚 CE 置高并保持 10uS 以上，同时延

迟 130uS 后发送数据；如果开启自动应答功能时，nRF24L01 在发送数据后就进入接收模式，接收应答信号。若收到应答，则表示此次通信成功；若没有收到应答，就自动重新发射该数据（自动重启功能需开启）。当发送成功后，IRQ 中断标志变低，通过 SPI 通知处理器。在一次发送成功后，如果发送堆栈中有数据而且使能为高时，则进入下一次发射；否则进入空闲模式。

在接收数据时，先将nRF24L01设置为接收模式，接着延迟130uS进入接收状态等待数据的到来。当接收检测到有效的地址和CRC时，就将数据包存储在接收堆栈中，同时中断标志位IRQ置低，通知处理器取数据。如果开启自动应答，接收方同时接入发射状态，回传应答信号。直到接收结束，便将使能关闭，进入空闲模式。

3.29.8 单片机串口硬件连接原理

神舟51开发板板载有两个2.4G无线数传模块NRF24L01 (或NRF24L01+)连接器。单片机的IO与NRF24L01无线模块相连，模拟SPI协议与NRF24L01通信。

第一块2.4G无线数传模块NRF24L01的硬件接口原理图如下图3-189所示：

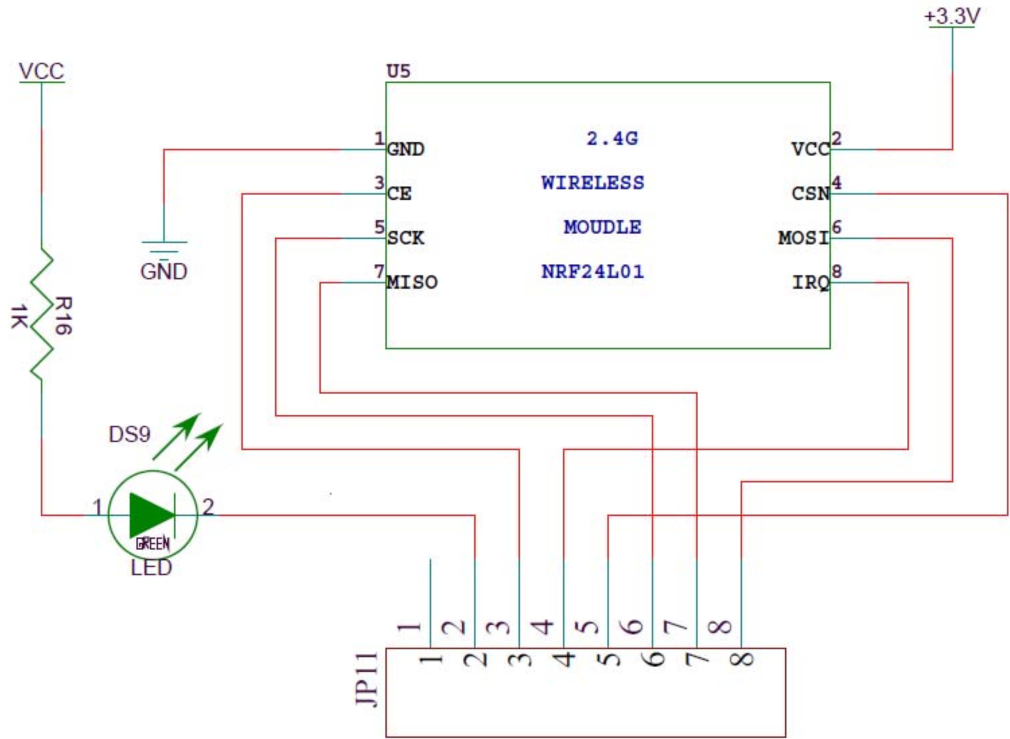


图 3-189 2.4G 无线模块硬件接口原理图 1

第二块2.4G无线数传模块NRF24L01的硬件接口原理图如下图3-190所示：

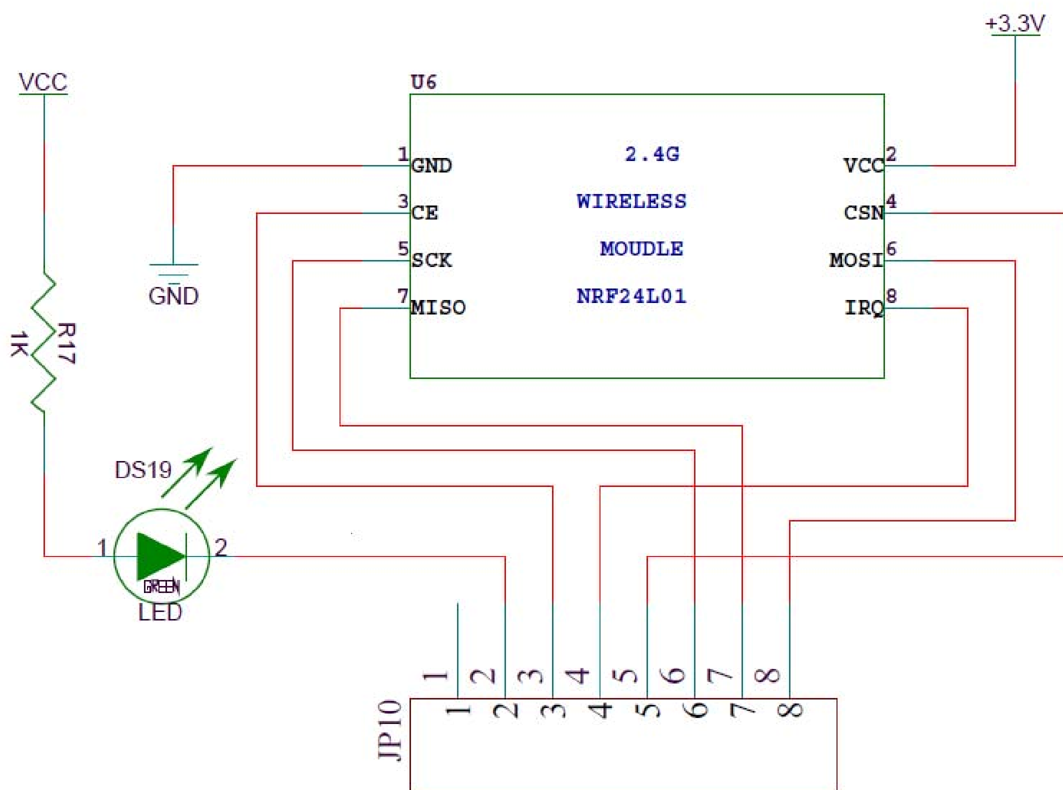


图 3-190 2.4G 无线模块硬件接口原理图 2

实验步骤如下：

1） 请将两个 2.4G模块安装在神舟 51 开发板的对应插座上。

2） 单片机烧录好固件。

3） 使用一根母到母交叉串口线连接好串口或者使用USB线连接到 51 开发板的USB座至电脑。

4） 打开电脑的超级终端工具，将波特率设置为 9600，无奇偶校验。

设置超级终端的方法如下：将神舟 51 开发板上的串口用以上任意一种方式与PC电脑连接以后，在PC上打开超级终端。打开方式为：Window下点击左下角的“开始”-->程序-->附件-->通讯-->超级终端，如下图 3-191 所示：

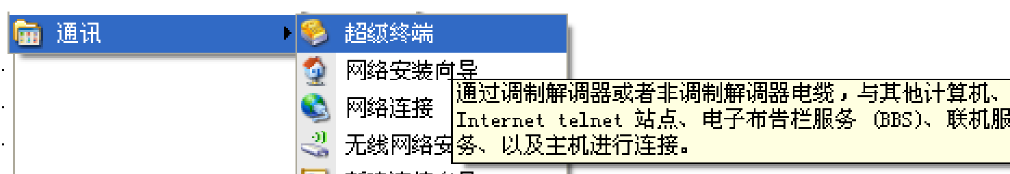


图 3-191 打开超级终端

按如下方式设置参数，如下图3-192：

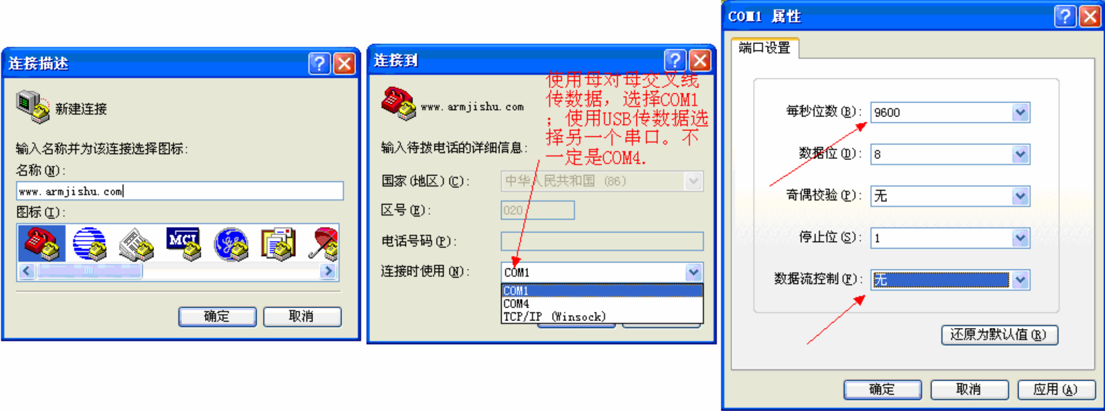


图 3-192 设置终端参数

开发板上电后可以看到模块的状态以及模块收发的数据内容，如果接收到的数据和发送的数据一致则说明实验成功，否则实验失败。如下图 3-193 正常发送与接收成功。



图 3-193 终端正常发送与接收成功

注意：晶振 11.0592MHz，波特率为 9600，使用串口或者 USB 转串口要检查 J21 和 J22 跳线跳帽的位置。

连接图如下图3-194所示：

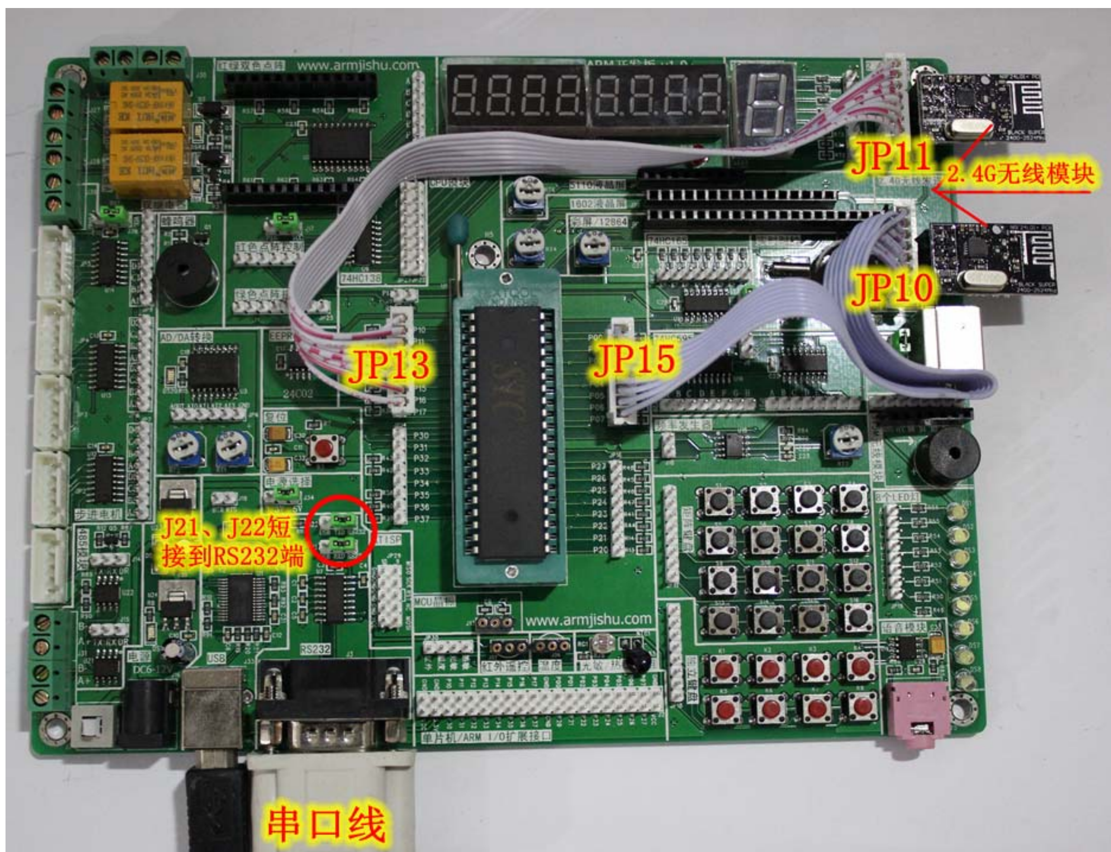


图 3-194 硬件连接实物图

3.29.9 例程 01 两块 2.4G无线数传模块测试实验

本实验在神舟 51 开发板上使用两块 2.4G 无线数传模块，检测是否板子和模块是否连接正确。将结果在单片机的串口输出。大概流程如下：

先向nRF24L01写入数据，然后把写入的数据读出，将写入和读出的数据进行对比。从而检测nRF24L01无线模块是否连接正常，通过串口打印连接的情况。一块块检测，测完一块再测另一块。

如果使用DB9串口座，则神舟51开发板的接线方法如下表3-130所示。

表3-130 硬件连接方法

单片机接口	插座 1	方式	插座 2	线缆	功能
P1	JP13(A 向左)	交叉	JP11(A 向左)	8 芯排线	2.4G 无线模块 1 信号
P0	JP15(A 向左)	交叉	JP10(A 向左)	8 芯排线	2.4G 无线模块 2 信号
RXD	J21.RXD	跳帽	J21.RS232		单片机串口接收
TXD	J22.TXD	跳帽	J22.RS232		单片机串口发送
J3 DB9		串口线		母对母交叉	与 PC 电脑连接
实验现象：如果串口提示模块的状态正常，显示模块收发的数据内容，如果接收到的数据和发送的数据一致则说明实验成功，否则实验失败。					

如果使用板载的USB转串口芯片PL2303，则神舟51开发板的接线方法如下表3-131所示。

表3-131 硬件连接方法

单片机接口	插座 1	方式	插座 2	线缆	功能
P1	JP13(A 向左)	交叉	JP11(A 向左)	8 芯排线	2.4G 无线模块 1 信号
P0	JP15(A 向左)	交叉	JP10(A 向左)	8 芯排线	2.4G 无线模块 2 信号
RXD	J21.RXD	跳帽	J21.USB		单片机串口接收
TXD	J22.TXD	跳帽	J22.USB		单片机串口发送
J33 USB				USB 线缆	与 PC 电脑连接
实验现象：如果串口提示模块的状态正常，显示模块收发的数据内容，如果接收到的数据和发送的数据一致则说明实验成功，否则实验失败。					

3.29.10 例程 02　两块 2.4G无线数传模块通信实验

本实验在神舟 51 开发板上使用两块 2.4G 无线数传模块通信，将结果在单片机的串口输出。

本程序在上一个程序的基础上，将 nRF24L01 模块 1 设置为接收模式，将 nRF24L01 模块 2 设置为发送模式。通过模块 2 周期性的发送变化的数据，并在串口显示发送内容和是否成功；模块 1 作为接收方等待并接收数据，并在串口显示接收到的内容。

3.30 5110 液晶屏

3.30.1 5110 液晶屏简介

液晶显示屏的英文名是 Liquid Crystal Display，简称 LCD。在日常生活中，我们对液晶显示器并不陌生。液晶显示模块已作为很多电子产品的通过器件，如在计算器、万用表、电子表及很多家用电子产品中都可以看到。液晶显示器(LCD)具有功耗低、体积小、重量轻、超薄等许多其它显示器无法比拟的优点,近几年来被广泛用于单片机控制的智能仪器、仪表和低功耗电子产品中。在袖珍中应用也越来越广泛。液晶显示技术近几年来发展很快，各种规格的 LCD 显示班名目繁多，其专用驱动芯片也都相互配套，使 LCD 在控制和议表系统中广泛应用提供了极大的方便。本章节我们介绍下 NOKIA5110 液晶 LCD，最早是 NOKIA5110 手机使用的屏幕。NOKIA5110 液晶 LCD 显示的每一个点只有透明和不透明两种状态，但是其背光根据 LED 灯颜色的不同可以有不同颜色的背光。下图 3-195 为 5110 点阵型液晶实物图：



图 3-195　5110 点阵型液晶实物图

LCD 可分为段位式 LCD、字符式 LCD 和点阵式 LCD。其中,段位式 LCD 和字符式 LCD 只能用于字符和数字的简单显示,不能满足图形曲线和汉字显示的要求;而点阵式 LCD 不仅

可以显示字符、数字,还可以显示各种图形、曲线及汉字,并且可以实现屏幕上下左右滚动,动画功能,分区开窗口,反转,闪烁等功能,用途十分广泛。而 NOKIA5110 液晶 LCD 属于一款点阵式 LCD。

3.30.2 5110 液晶屏的原理和特点

- 84x48 的点阵 LCD，可以显示 4 行汉字，
- 采用串行接口与主处理器进行通信，接口信号线数量大幅度减少，包括电源和地在内的信号线仅有 9 条。支持多种串行通信协议（如 AVR 单片机的ＳＰＩ、MCS51 的串口模式 0 等），传输速率高达 4Mbps，可全速写入显示数据，无等待时间。
- 可通过导电胶连接模块与印制版，而不用连接电缆，用模块上的金属钩可将模块固定到印制板上，因而非常便于安装和更换。
- LCD 控制器／驱动器芯片已绑定到 LCD 晶片上，模块的体积很小。
- 采用低电压供电,正常显示时的工作电流在 200μA 以下,且具有掉电模式。LPH7366 的这些特点非常适合于电池供电的便携式通信设备和测试设备中
- 性价比高，LCD1602 可以显示 32 个字符，而 Nokia5110 可以显示 15 个汉字，30 个字符。Nokia5110 裸屏仅 8.8 元，LCD1602 一般 15 元左右，LCD12864 一般 50~70 元。
- 接口简单，仅四根 I/O 线即可驱动，LCD1602 需 11 根 I/O 线，LCD12864 需 12 根。
- 速度快，是 LCD12864 的 20 倍，是 LCD1602 的 40 倍。
- Nokia5110 工作电压 3.3V，正常显示时工作电流 200uA 以下，具有掉电模式，适合电池供电的便携式移动设备。

利用 PC 上的 16×16 点阵汉字库，提取后将点阵文件存入 ROM，直接利用 PC 中汉字内码作为单片机系统的编码（不再形成新的汉字编码）。

在数字电路中，所有的数据都是以 0 和 1 保存的，对 LCD 控制器进行不同的数据操作，可以得到不同的结果。对于显示英文操作，由于英文字母种类很少，只需要 8 位（一字节）即可。而对于中文，常用却有 6000 以上，将 ASCII 表的高 128 个很少用到的数值以两个为一组来表示汉字，即汉字的内码。而剩下的低 128 位则留给英文字符使用，即英文的内码。

那么，得到了汉字的内码后，还仅是一组数字，那又如何在屏幕上去显示呢？这就涉及到文字的字模，字模虽然也是一组数字，但它的意义却与数字的意义有了根本的变化，它是用数字的各位信息来记载英文或汉字的形状，如英文的'A'在字模的记载方式如图 3-196 所示：

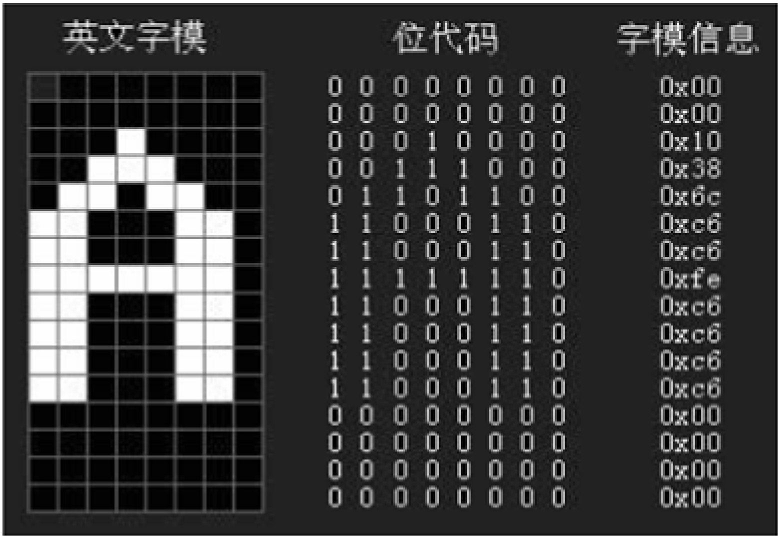


图 3-196 “A”字模图

而中文的“你”在字模中的记载却如图 3-197 所示：

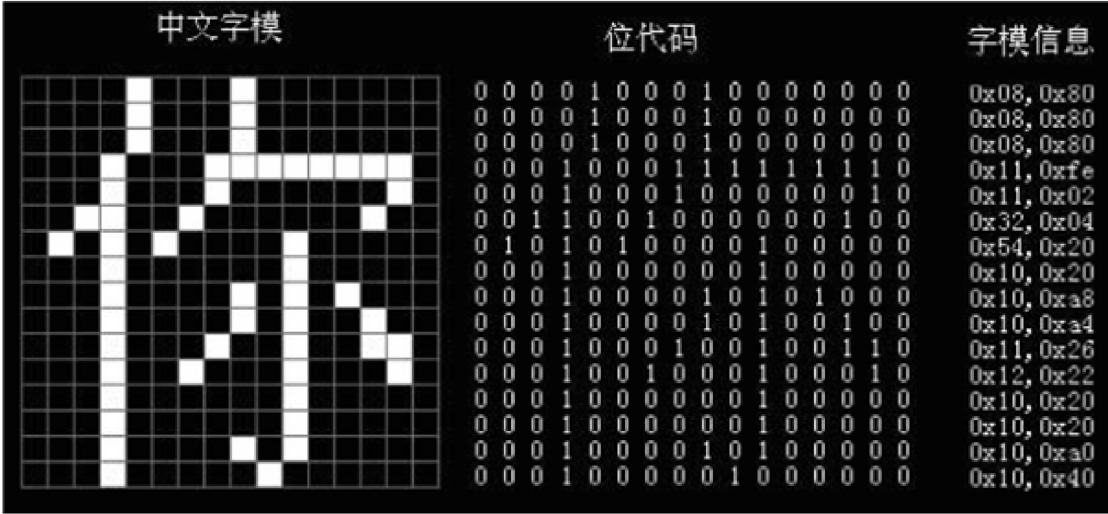


图 3-197“你”字模图

这些内码我们可以通过一个字模生成工具来得到，在下面的小节里面我们会详细的介绍。

3.30.3 5110 液晶屏连接方式

我们的 5110LCD 属于第 3 种连接方式，直接通过导电橡胶连接，常用液晶模块连接方式及其适用范围：

➢ 金属插脚

金属引脚可直接焊接在 PCB（印刷线路板）上， 连接可靠，抗震动强，脚间距受限制 适用于音响产品，电表等。

➢ 斑马纸(热封)

柔软性连接，组装不方便。 用于薄型产品的连接。 适用于计数器、寻呼机、电子记事簿等。

➢ 导电橡胶

成本较低，组装方便， 是较多采用的连接方式 成本较低。 适用于手表，游戏机，时钟，电话等。

3.30.4 5110 液晶屏管脚分析

5110 屏引脚图如下图 **3-198** 所示：

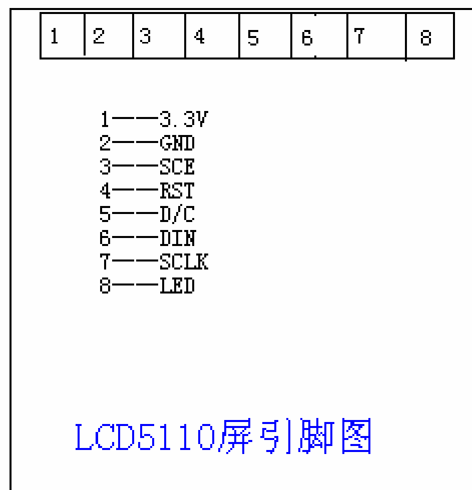


图 3-198 5110 屏引脚图

VCC——模块电源，范围为 2.7-3.3V，强烈建议不要直接接到 5V 上去。
GND——模块电源地及背光地。
SCE——模块使能引脚，允许输入数据，低电平有效。
RST——模块复位引脚，复位模块，应用于初始化模块，低电平有效。
D/C——模式选择，选择命令/地址或输入数据。
DIN——串行数据引脚，串行输入数据线。
SCLK——串行时钟引脚，时钟信号，0~4.0Mbit/s。
LED——背光电源，电压 5-8V

3.30.5 5110 液晶屏字模生成方法

安装光盘中的“LCD 字模 III 软件”，安装完后打开界面如下图 3-199，生成汉字或图片字库有五个步骤，下面一一介绍。



图 3-199 LCD 字模 III 软件

第一步，选择字体。根据需要选择合适的或者喜欢的字体。如下图 3-201：



图 3-200 选择字体

第二步，设置参数体。按如下方式设置参数。如下图 3-201、3-202、3-203：

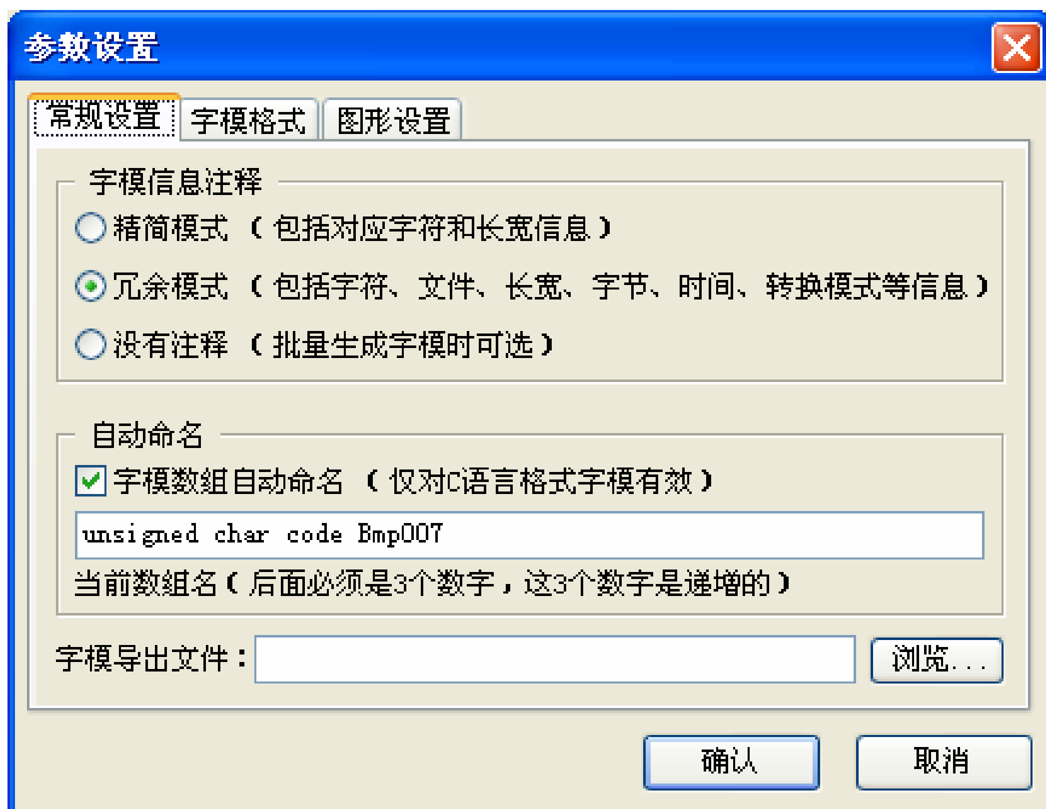


图 3-201 设置参数体

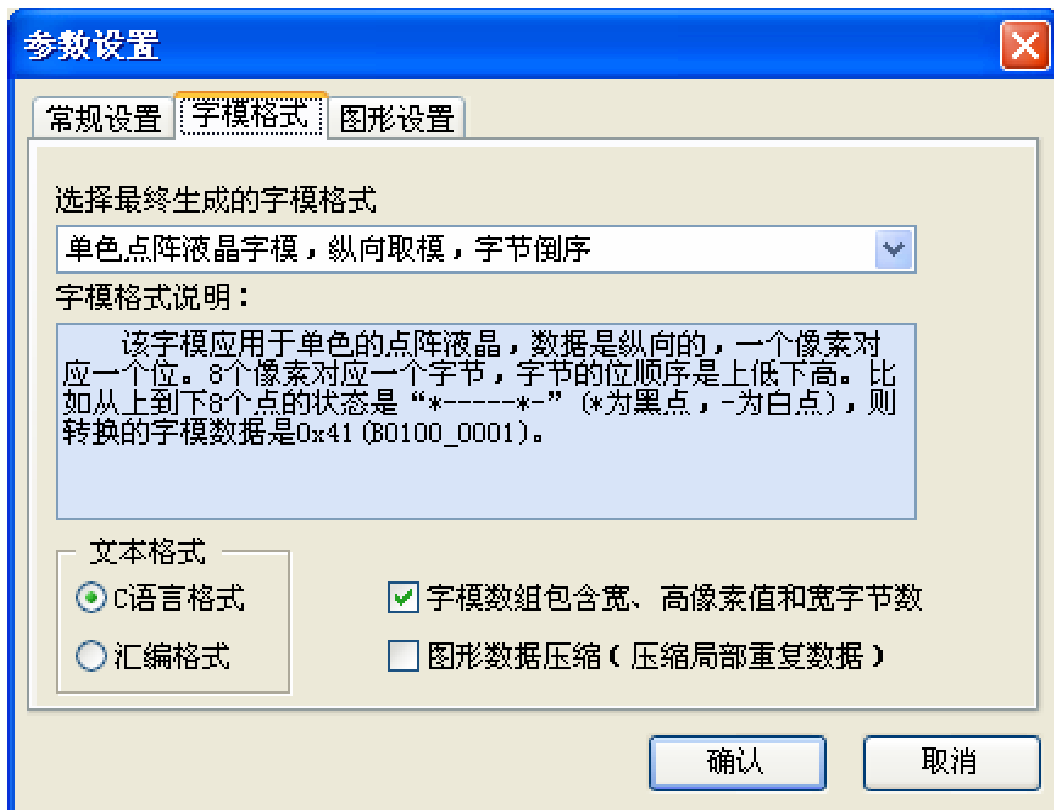


图 3-202 设置参数体

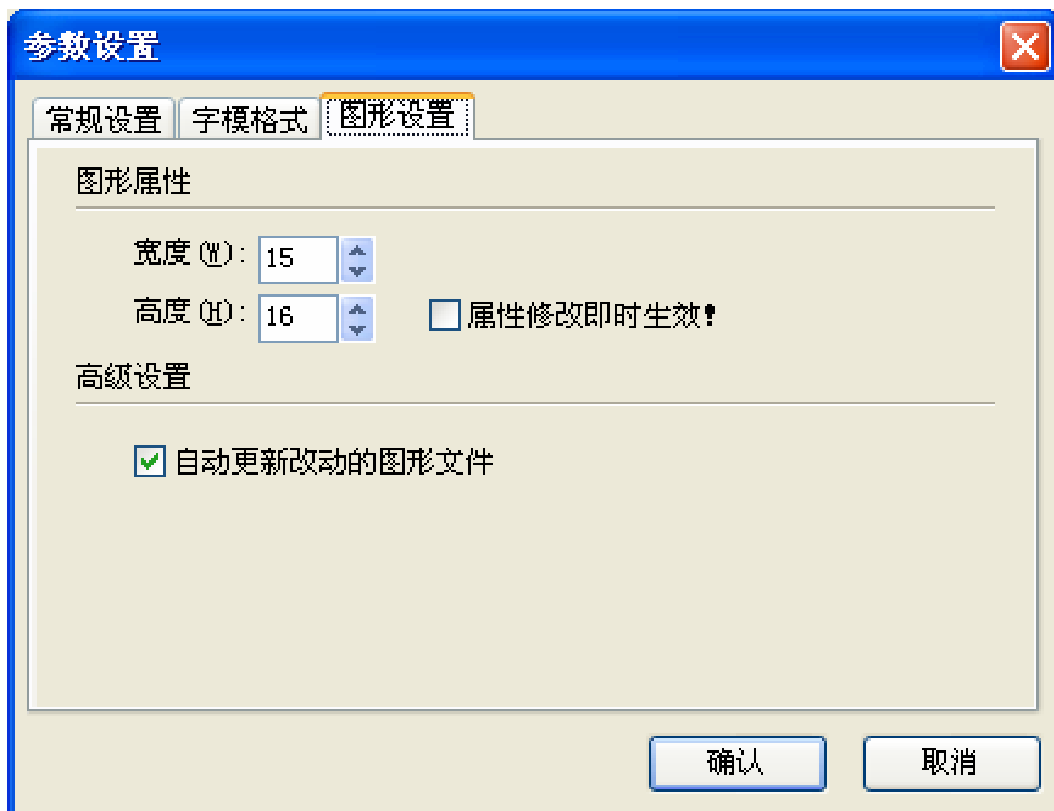


图 3-203 设置参数体

第三步，输入待转换的字符串。记得将右下角的高度设置为 8 的整数倍。如下图 3-204：

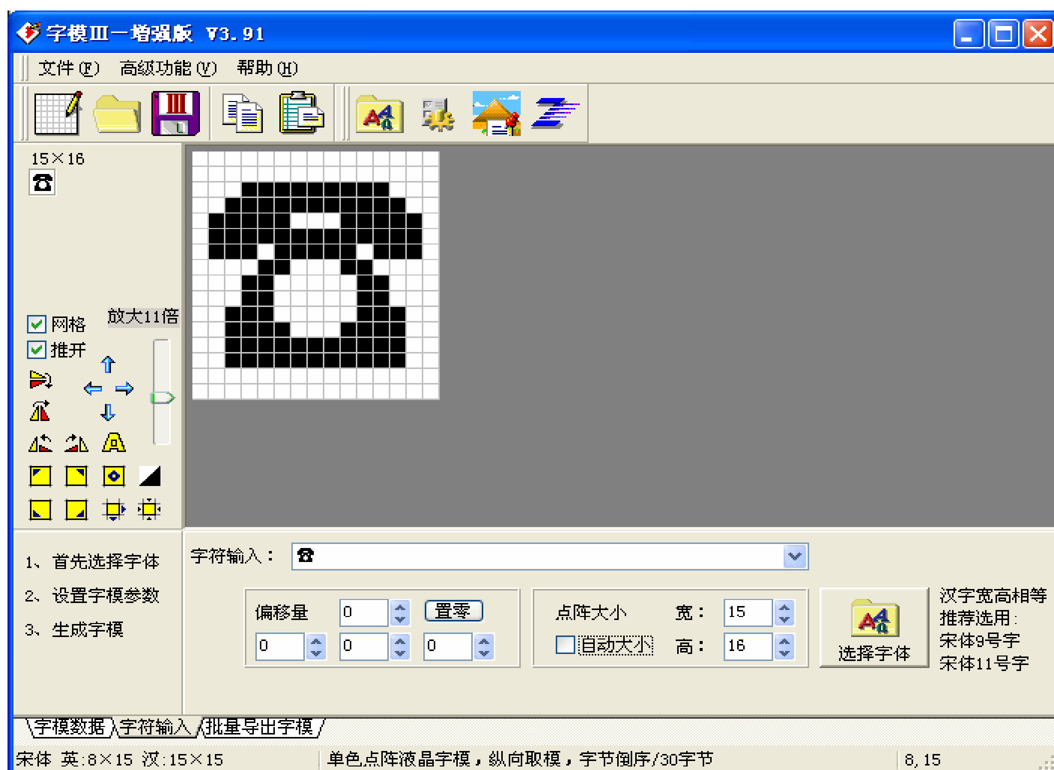


图 3-204 输入待转换的字符串

第四步，修改点阵。如果对点阵不满意，可以点击对应的点对其修改。也可以按这种方式设计自己的特殊符号图标，如下图 3-205 所示。

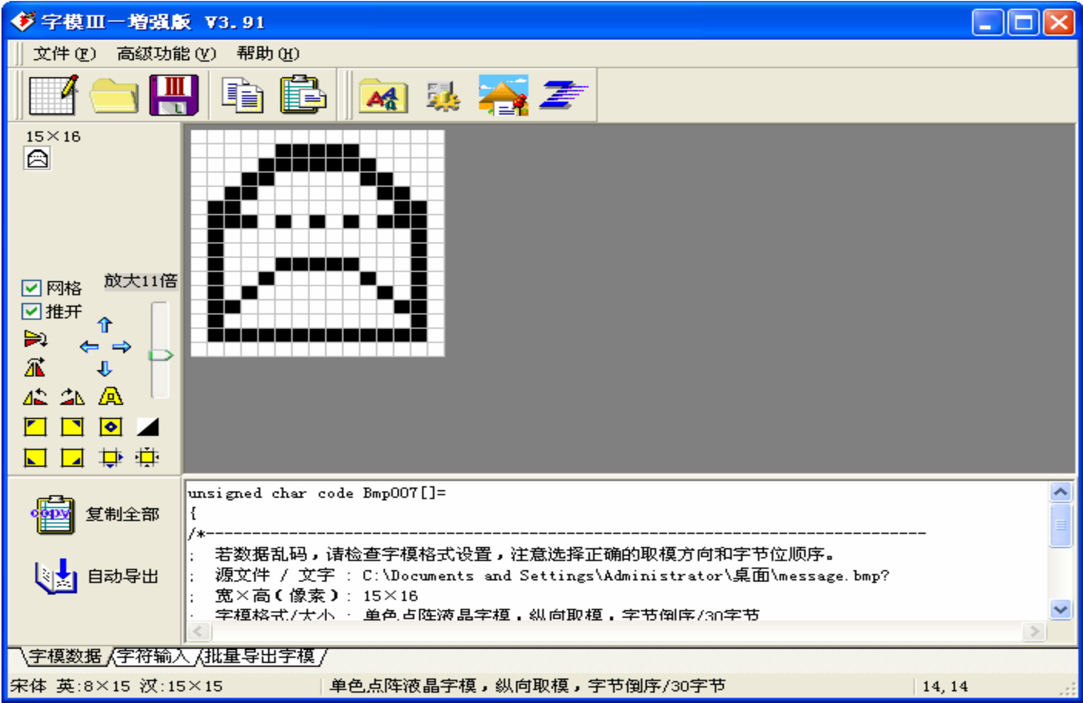


图 3-205 修改点阵

第五步，开始转换。对对点阵修改满意后，就可以点击生成字模数据，如下图 3-206：

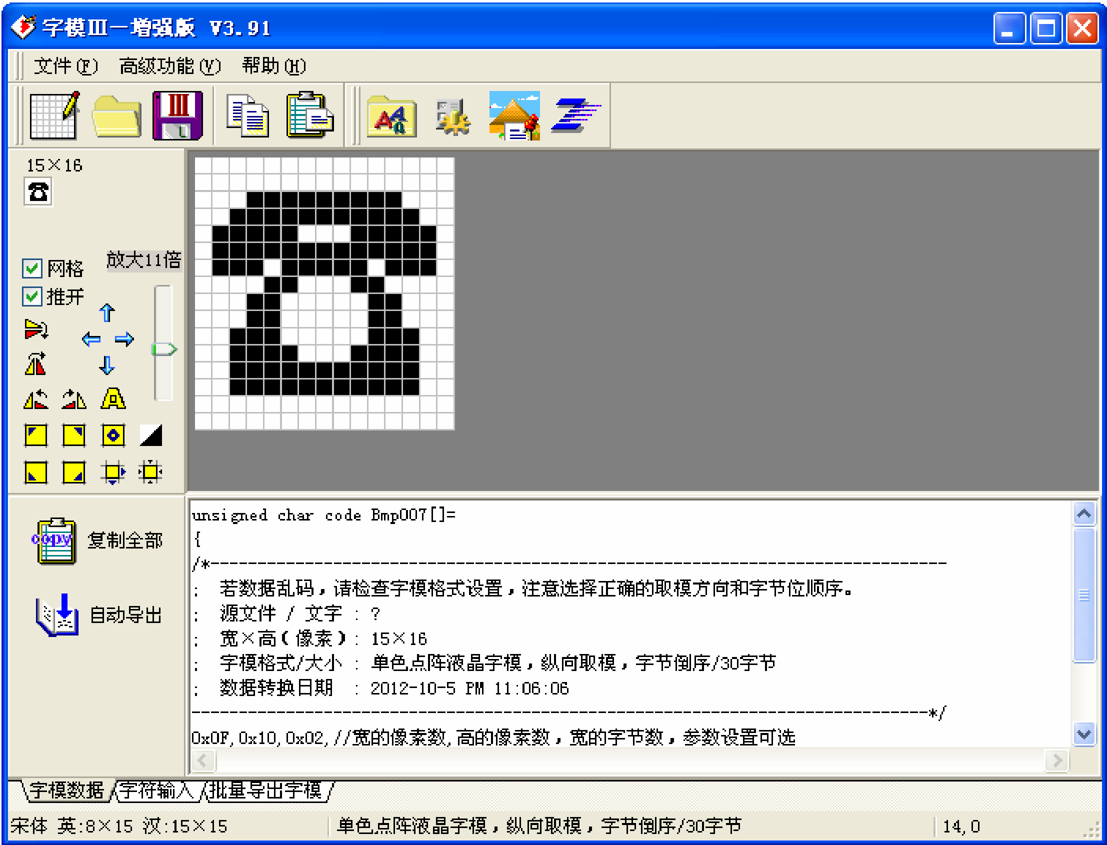


图 3-206 开始转换

3.30.6 如何控制 5110 液晶屏

NOKIA5110 液晶 LCD 的驱动芯片 PCD8544，所以对 NOKIA5110 的访问实质就是对 PCD8544 的访问。PCD8544 内置 48*84 位用于显示的 RAM 矩阵，数据以字节为单位写入到 PCD8544 的 48*84 位显示数据 RAM 矩阵，列是通过地址指针寻址的，地址范围为：X 0~83（1010011）；一共 48 个点阵行，每 8 个点阵行为一个字节行（以下简称"行"），所以一共有 6 行 Y 为 0~5（101）。地址不充许超出这个范围。在垂直寻址模式（V=1），Y 地址在每个字节之后递增。经最后的 Y 地址（Y=5）之后，Y 绕回 0，X 递增到下一列的地址。在水平寻址模式（V=0），X 地址在每个字节之后递增 ,经最后的 X 地址 （X=83）之后， X 绕回 0， Y 递增到下一行的地址。 经每一个最后地址之后 X=83，Y=5)，地址指针绕回地址（X=0，Y=0）。

LCD 的驱动芯片 PCD8544 的对外接口为 SPI 接口，下面我们来介绍 SPI 模式 。

SPI 总线通信基于主-从配置。它有以下 4 个信号：

MOSI: 主器件数据输出,从器件数据输入（主出/从入）

MISO: 主器件数据输入,从器件数据输出（主入/从出）

SCK: 时钟信号,由主器件产生（串行时钟）

SS（CS）: 从器件使能信号,由主器件控制 （从属选择）

5110 液晶显示屏（PCD8544 芯片），是通过发送指令和写入数据 RAM 来控制和显示数据的。

指令格式分为两种模式：

1、如果 D/C（模式选择）置为低(为 0)，即位变量 D/C= 0，为发送指令模式，那么接下来发送的 8 位字节解释为命令字节。

2、如果 D/C 置为高，即 D/C = 1; 为写入数据 RAM 模式，接下来的字节将存储到显示数据 RAM。

注意：

1、每一个数据字节存入之后， 地址计数自动递增。 在数据字节最后一位期间会读取 D/C 信号的电平。

2、每一条指令可用任意次序发送到 PCD8544。首先传送的是字节的 MSB（高位）

前面我们知道 5110 液晶显示屏（PCD8544 芯片），是通过发送指令和写入数据 RAM 来控制和显示数据的，那么我们现在看下 PCD8544 芯片的指令集，了解下它的作用与使用，下表 3-132 为该芯片的指令集，深色部分表 3-133 为符号说明。

表 3-132 指令集

指令	D/C	命令字								描述
		DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	
（H=0 or 1）										
NOP	0	0	0	0	0	0	0	0	0	空操作
功能设置	0	0	0	1	0	0	PD	V	H	掉电控制：进入模式：扩展指令设置（H）
写数据	1	D7	D6	D5	D4	D3	D2	D1	D0	写数据到显示 RAM

（H=0）										
保留	0	0	0	0	0	0	1	X	X	不可使用
显 示 控制	0	0	0	0	0	1	D	0	E	设置显示配置
保留	0	0	0	0	1	X	X	X	X	不可使用
设 置 RAM 的 Y 地址	0	0	1	0	0	0	Y2	Y1	Y0	设 置 RAM 的 Y 地址 0≤Y≤5
设 置 RAM 的 X 地址	0	1	X6	X5	X4	X3	X2	X1	X0	设 置 RAM 的 X 地址 0≤X≤83
(H=1)										
保留	0	0	0	0	0	0	0	0	1	不可使用
	0	0	0	0	0	0	0	1	X	不可使用
	0	0	0	0	0	0	1	TC1	TC0	设置温度系数（TCX）
	0	0	0	0	0	1	X	X	X	不可使用
	0	0	0	0	1	0	BS2	BS1	BS0	设置偏置系统 BSX
	0	0	1	X	X	X	X	X	X	不可使用
	0	1	VOP6	VOP5	VOP4	VOP3	VOP2	VOP1	VOP0	写 VOP 到寄存器

表 3-133 符号说明

BIT	0	1
PD	芯片是活动的	芯片处于掉电模式
V	水平寻址	垂直寻址
H	使用基本指令集	使用扩展指令集
D and E 00 10 01 11	 显示空白 普通模式 开所有显示段 反转换象模式	
TC1and TC0 00 01 10 11	 VLCD 温度系数 0 VLCD 温度系数 1 VLCD 温度系数 2 VLCD 温度系数 3	

结合表 1 和表 2, 指令集不难读懂。现在以红色横线所标识的指令来举例说明：

功能设置指令：

首先，D/C 为 0, 表示现在是指令模式，然后从表 2 看：
若要“使用基本指令集”，则 PD = 0, V = 0, H = 0，那么相应地 DB7~DB0 分别为 00100000b，即 0x20。也就是说，发送 0x20 就能设置液晶“使用基本指令集”。
同理，使用“使用扩展指令集”，指令值为 0x21

显示控制：
显示模式有 4 种，分别是：显示空白，普通模式，开所有显示段，反转映象模式。这些模式分别使用 D and E 来组合控制。
这样，我们控制显示为“普通模式”, 那么 D 要为 1，E 要为 0, 这时 DB7~DB0 分别为 0000 1100b，即指令值为 0x0C。

写入数据指令：
写入数据，首先 D/C 必须为 1，表示现在是写入数据 RAM 模式，DB7~DB0 就是相应的数据值。

到这里，我们知道了 5110 液晶屏是使用这样的指令集来控制和显示数据的。那么，我们如何把指令值写进去呢？

要向液晶屏写入数据，我们需要通过模拟一个串行总线协议来写入数据。 先来看看传送一个字节（指令）的时序图，如下图 3-207：

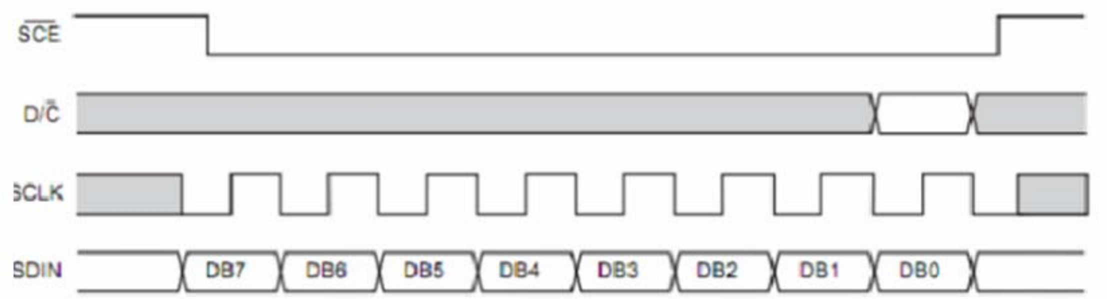


图 3-207 串行总线协议——传送一个字节

从时序图 1 看出：
1. SCE 片选为 0 时开始发送数据。
2. 时钟信号 SCLK 仅在 SCE 片选为 0 时有效。
3. SDIN 数据输入，需要在时钟信号 SCLK 有效时开始发送数据，且在 SCLK 的正边缘取样
注意：数据是从高位开始发送的。
知道了传送数据的时序，我们需要使用程序来模拟这个时序，从而发送指令控制液晶屏。

3.30.7 硬件连接原理

神舟51开发板板载有一个5110液晶屏模块接口连接器。单片机的IO与5110液晶屏模块相连。如下图3-208为硬件原理图

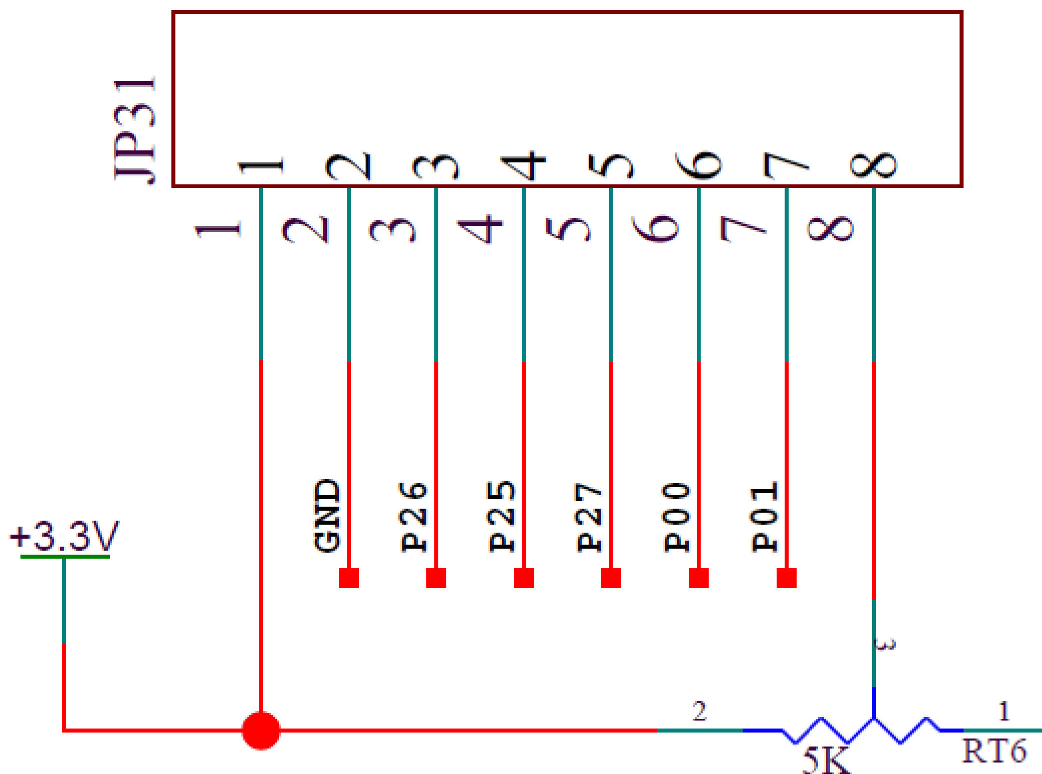


图 3-208　 5110 屏原理图

其中 5110 的第 8 脚 LED 是 5110 液晶的背光调节，如果液晶需要背光与调节亮度的话，可以调节 8 脚上的电位器，来达到最理想的显示效果。

3.30.8 例程 01 NOKIA5110 液晶LCD显示英文

代码如下：

```
/*****************************************************************
*  例程：NOKIA5110 液晶 LCD 显示英文
*  作者：www.armjishu.com
*  版本：v1.0
*  内容：通过 51 单片机 IO 管脚与 NOKIA5110 液晶 LCD 通信，实现英文字符串显示
*  现象：NOKIA5110 液晶 LCD 的第一行显示字符串" armjishu.com "第二行显示字
*         符串" ARMJISHU.COM "，前两行白底黑底和黑底白字交替的显示，第 3 行
*         至第 6 行移屏显示英文字符。
*****************************************************************/
/*  包含头文件  */
#include <reg51.h>
#include "delay.h"
#include "fonts.h"
sbit LCD_SCE = P2^6;        //  片选
sbit LCD_DC    = P2^7;       // 1 写数据，0 写指令
sbit LCD_RST = P2^5;        //  复位,0 复位
```

```
sbit LCD_DIN = P0^0;        //  数据
sbit LCD_CLK = P0^1;       //  时钟
#define LCD_DATA      1
#define LCD_CMD        0
/*************************************************************/
/*                         函数声明                          */
/*************************************************************/
void LCD_write_byte(unsigned char cdata, unsigned char command);
void LCD_set_XY(unsigned char X, unsigned char Y);
void LCD_clear(void);
void LCD_init(void);
void LCD_write_char(unsigned char cdata,unsigned char mode);
void LCD_write_String(unsigned char *str, unsigned char mode);
/*-----------------------------------------------
              主函数
-----------------------------------------------*/
void main(void)
{
     unsigned char i,j=0;
     unsigned char MyStr1[] = " armjishu.com ";
     unsigned char MyStr2[] = " ARMJISHU.COM ";
     LCD_init();       //初始化 LCD 模块
     LCD_clear();     //清屏幕
     while(1)
     {
          //LCD 的第一行显示字符串" armjishu.com "
          LCD_set_XY(0,0);
          LCD_write_String(MyStr1,0);
          //LCD 的第二行显示字符串" ARMJISHU.COM "
          LCD_set_XY(0,1);
          LCD_write_String(MyStr2,1);
          //第 3 行至第 6 行移屏显示英文字符。
          LCD_set_XY(0,2);

          for(i=0;i<4*14;i++)
          {
               LCD_write_char(j+i+' ',0);
          }
          if(j>(('z' - ' ')-(4*14)))
          {
               j=0;
          }
          else
          {
```

```
            j++;
        }

        DelayMs(200);
        DelayMs(200);
        //前两行白底黑底和黑底白字交替的显示
        LCD_set_XY(0,0);
        LCD_write_String(MyStr1,1);
        LCD_set_XY(0,1);
        LCD_write_String(MyStr2,0);
        DelayMs(200);
        DelayMs(200);
    }
}
/*-----------------------------------------------------------------
LCD_write_byte: 使用 SPI 接口写数据到 LCD
输入参数：data：写入的一个字节数据；
command ：写数据/命令选择；
-----------------------------------------------------------------*/
void LCD_write_byte(unsigned char cdata, unsigned char command)
{
    unsigned char i;
    LCD_SCE=0;
    LCD_DC=command;
    for(i=0;i<8;i++)
    {
        if(cdata&0x80)
        {
            LCD_DIN=1;
        }
        else
        {
            LCD_DIN=0;
        }
        cdata=cdata<<1;
        LCD_CLK=0;
        LCD_CLK=1;
    }
    LCD_DC=1;
    LCD_SCE=1;
    LCD_DIN=1;
}
/*---------------------------------------------------
LCD_set_XY: 设置 LCD 坐标函数
```

```
输入参数：X(column 列)：0－83    Y(page 页)：0－5
--------------------------------------------------*/
void LCD_set_XY(unsigned char X, unsigned char Y)
{
    LCD_write_byte(0x40 | Y, LCD_CMD);      // Y(page 页)
    LCD_write_byte(0x80 | X, LCD_CMD);      // X(column 列)
}
/*--------------------------------------------------
LCD_clear: LCD 清屏函数
--------------------------------------------------*/
void LCD_clear(void)
{
    unsigned char t;
    unsigned char k;
    LCD_set_XY(0,0);
    for(t=0;t<6;t++)
    {
        for(k=0;k<84;k++)
        {
            LCD_write_byte(0x00,LCD_DATA);
        }
    }
}
/*-----------------------------------------------
LCD_init: 5110LCD 初始化
-----------------------------------------------*/
void LCD_init(void)
{
    LCD_RST=0;
    DelayMs(10);
    LCD_RST=1;
    LCD_write_byte(0x21, LCD_CMD);  //  使用扩展命令设置 LCD 模式
    LCD_write_byte(0xc8, LCD_CMD);  //  设置偏置电压
    LCD_write_byte(0x06, LCD_CMD);  //  温度校正
    LCD_write_byte(0x13, LCD_CMD);  // 1:48
    LCD_write_byte(0x20, LCD_CMD);  //  使用基本命令
    LCD_write_byte(0x0c, LCD_CMD);  //  设定显示模式，正常显示
    LCD_clear();                    //  清屏
}
/*-----------------------------------------------------------------------
LCD_write_char:  显示英文字符
输入参数：data：显示的字符；
          mode   为 0 时白底黑字，为 1 时黑底白字
-----------------------------------------------------------------------*/
```

```
void LCD_write_char(unsigned char cdata,unsigned char mode)
{
    unsigned char line;
    cdata -= 32;
    for (line=0; line<6; line++)
    {
        if(mode)
        {
            LCD_write_byte(~font6x8[cdata][line], LCD_DATA);
        }
        else
        {
            LCD_write_byte(font6x8[cdata][line], LCD_DATA);
        }
    }
}
/*-----------------------------------------------------------------------
LCD_write_char: 英文字符串显示函数
输入参数：*str：英文字符串指针
          mode   为 0 时白底黑字，为 1 时黑底白字
-----------------------------------------------------------------------*/
void LCD_write_String(unsigned char *str, unsigned char mode)
{
    while (*str)
    {
        LCD_write_char(*str,mode);
        str++;
    }
}
```

本实验是通过 51 单片机的 IO 管脚与 NOKIA5110 液晶 LCD 通信，实现英文字符串显示。NOKIA5110 液晶 LCD 的第一行显示字符串" armjishu.com "第二行显示字符串" ARMJISHU.COM "，前两行白底黑底和黑底白字交替的显示，第 3 行至第 6 行移屏显示英文字符。

由于NOKIA5110液晶LCD接口已经连接到了51单片机的IO管脚，所以本实验无需额外的连线，只需要安装好NOKIA5110液晶LCD即可。

实验效果图如下图3-209、3-210、3-211所示：



图 3-209 实验效果图 1



图 3-210 实验效果图 2

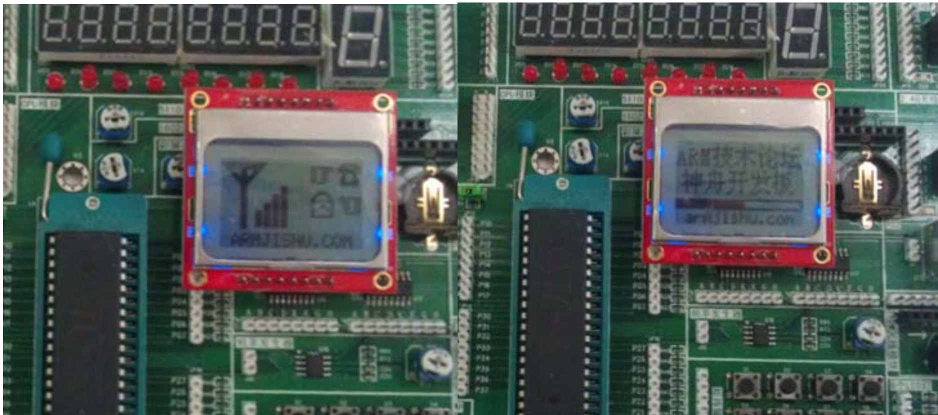


图 3-211 实验效果图 3

3.30.9 更多有关 5110 液晶屏显示等更多例程

更多 5110 液晶屏相应的例程可通过北京航空航天大学出版社下载专区下载，如下表 3-134：

表 3-134 5110 液晶屏更多丰富例程（含详细注释和文档分析）

序号	例程功能
例程 01	NOKIA5110 液晶 LCD 显示英文
例程 02	NOKIA5110 液晶 LCD 显示中文与英文
例程 03	NOKIA5110 液晶 LCD 显示进度条
例程 04	NOKIA5110 液晶 LCD 显示图标

3.31 TFT彩色液晶屏

3.31.1 术语解释

将普通图像的信息数字化后，即得到数字图像，最后转换为合适计算机处理的数字。LCD 控制器则被用于将数字图象传输到 LCD 显示屏进行显示。所以，如果需要理解 LCD 控制器的工作原理，有必要了解一些图象基础知识

1）帧

显示屏所显示的一幅完整画面就是一个帧。

2）分辨率

分辨率有很多种，例如打印分辨率、影像分辨率等。此处仅仅就 LCD 显示器的分辨率进行阐释。分辨率是指显示器所能显示点数的多少，可以把整个显示器想象成是一个棋盘，而分辨率就是棋盘经线和纬线交叉点的数目。由于屏幕上的点、线和面都是由点组成的，所以显示器可以显示的点数越多，显示画面就越精细。

对于 LCD 显示器来说，象素的数目和分辨率在数值上相等，都等于屏幕上横向点个数和纵向点个数的乘积。比如 2.4 寸液晶屏的分辨率是 240*320 的，有 240*320=76800 这么多个点。

3）象素

“象素”是由图像和元素这两个单词的字母所组成的，是构成数字图象的最小单位。我们若把数字图象放大数倍，就会发现数字图象其实是由许多色彩相近的小方点所组成，这些小方点就是“象素”。 一个像素所能表达的不同颜色数取决于比特每像素(BPP)，这个最大数可以通过取二的色彩深度次幂来得到，它有可能由‘红-R’‘绿-G’‘蓝-B’三种颜色，即 RGB 组成，例如用 18 位来描述这三种颜色，其中每 6 位描述一种颜色，每种颜色可以呈现 2 的 6 次方种颜色，这样，由三种不同的颜色进行混搭，就可以变化出 2 的 18 次方这么多种不同的颜色；假如用 16 位来描述这三种颜色，就可以描述出 2 的 16 次方种颜色，即 65536 种颜色。

3.31.2 TFT彩屏硬件原理简介

LCD，即液晶显示器，因为其功耗低、体积小，承载的信息量大，因而被广泛用于信息输出、与用户进行交互，目前仍是各种电子显示设备的主流。

TFT就是“Thin Film Transistor”的简称，一般代指薄膜液晶显示器，而实际上指的是薄膜晶体管（矩阵）——可以“主动的”对屏幕上的各个独立的像素进行控制。对于图象产生的基本原理为：显示屏由许多可以发出任意颜色的光线的像素组成，主要控制各个像素显示相应的颜色就可以达到目的。在TFT LCD中一般会采用背光技术，为了能精确的控制每一个像素的颜色和亮度就需要在每一个想色之后安装一个类似百叶窗的开关，当“百叶窗”打开时光线就可以透射过来，而“百叶窗”关上之后，光线就无法透射。

图像数据的像素点由红(R)、绿(G)、蓝(B)三原色组成，三原色根据其深浅程度被分为0~255个级别，它们按不同比例的混合可以得出各种色彩。如R：255，G255，B255混合后为白色。根据描述像素点数据的长度，主要分为8、16、24及32位。如以8位来描述的像素点可表示28=256色，16位描述的为216=65536色，称为真彩色，也称为64K色。实际上受人眼对颜色的识别能力的限制， 16位色与12位色已经难以分辨了。

因为51单片机和STM32内部没有集成专用的液晶屏和触摸屏的控制接口，所以在显示面板中应自带含有这些驱动芯片的驱动电路(液晶屏和触摸屏的驱动电路是独立的)，51单片机和STM32芯片通过驱动芯片来控制液晶屏和触摸屏。以神舟51+ARM开发板2.4寸液晶屏(240*320)为例，它使用ILI9320或SSD1289或者8352型号的控制芯片来控制液晶屏，通过XPT2046芯片控制触摸屏。

液晶屏的控制芯片内部结构相对比较复杂，液晶屏的彩屏驱动电路最主要的是位于中间

GRAM(Graphics RAM)，可以理解为显存。GRAM 就好比是一个彩屏数据缓冲区，我们可以把大批的显示内容以显示矩阵的形式写到这个 GRAM 里，让彩屏 LCD 来读取 GRAM 里的数据再由彩屏驱动芯片显示到显示屏上，随着 GRAM 逐渐丰富和完善，除了显示矩阵以外还放很多的命令，GRAM 中每个存储单元都对应着液晶面板的一个像素点。它右侧的各种模块共同作用把 GRAM 存储单元的数据转化成液晶面板的控制信号，使像素点呈现特定的颜色，而像素点组合起来则成为一幅完整的图像。到现在，这些驱动/控制电路以及 GRAM 都合起来放在一片芯片中，统一被称为驱动 IC，这个驱动 IC 就是上一段我们所说的 8352 或者 ILI9320 或者 SSD1289 的驱动 IC 芯片。

3.31.3 液晶显示原理剖析

TFT就是“Thin Film Transistor”的简称，一般代指薄膜液晶显示器，而实际上指的是薄膜晶体管（矩阵）——可以“主动的”对屏幕上的各个独立的像素进行控制。对于图象产生的基本原理为：显示屏由许多可以发出任意颜色的光线的像素组成，主要控制各个像素显示相应的颜色就可以达到目的。在TFT LCD中一般会采用背光技术，为了能精确的控制每一个像素的颜色和亮度就需要在每一个想色之后安装一个类似百叶窗的开关，当“百叶窗”打开时光线就可以透射过来，而“百叶窗”关上之后，光线就无法透射。

图像数据的像素点由红(R)、绿(G)、蓝(B)三原色组成，三原色根据其深浅程度被分为0~255个级别，它们按不同比例的混合可以得出各种色彩。如R：255，G255，B255混合后为白色。根据描述像素点数据的长度，主要分为8、16、24及32位。如以8位来描述的像素点可表示28=256色，16位描述的为216=65536色，称为真彩色，也称为64K色。实际上受人眼对颜色的识别能力的限制， 16位色与12位色已经难以分辨了。

神舟51+ARM开发板上配带的2.4寸LCD屏，LCD屏为320x240分辨率，ILI9320液晶控制器液晶控制器自带显存，其总大小为自带显存，其总大小为172800（240*320*18/8）字节。

神舟51+ARM开发板支持2.4寸的ILI9320或ILI9328的TFT LCD，在本例程中，我们以ILI9320控制器进行简单的介绍。

ILI9320采用的是16位控制模式，以16位描述的像素点。按照标准格式，16位的像素点的三原色描述的位数为R：G：B =5：6：5，描述绿色的位数较多是因为人眼对绿色更为敏感

ILI9320最高能够控制18位的LCD，但为了数据传输简便，我们采用它的16位控制模式，以16位描述的像素点。按照标准格式，16位的像素点的三原色描述的位数为R：G：B =5：6：5，描述绿色的位数较多是因为人眼对绿色更为敏感。

ILI9320控制模块的16位数据与显存间的对应关系如图3-212所示：

ILI9320控制模块的16位数据与显存间的对应关系如下图3-213所示：

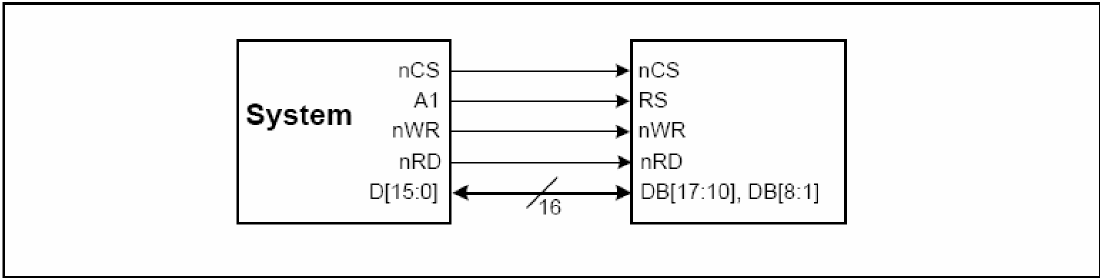


图3-212 ILI9320控制模块的16位数据与显存间的对应关系

TRI	DFM	16-bit MPU System Interface Data Format
0	*	system 16-bit interface (1 transfers/pixel) 65,536 colors 1st Transfer: DB17 DB16 DB15 DB14 DB13 DB12 DB11 DB10 DB8 DB7 DB6 DB5 DB4 DB3 DB2 DB1 R5 R4 R3 R2 R1 R0 G5 G4 G3 G2 G1 G0 B5 B4 B3 B2 B1 B0
1	0	80-system 16-bit interface (2 transfers/pixel) 262,144 colors 1st Transfer: DB17 DB16 DB15 DB14 DB13 DB12 DB11 DB10 DB8 DB7 DB6 DB5 DB4 DB3 DB2 DB1 2nd Transfer: DB17 DB16 R5 R4 R3 R2 R1 R0 G5 G4 G3 G2 G1 G0 B5 B4 B3 B2 B1 B0
1	1	80-system 16-bit interface (2 transfers/pixel) 262,144 colors 1st Transfer: DB2 DB1 2nd Transfer: DB17 DB16 DB15 DB14 DB13 DB12 DB11 DB10 DB8 DB7 DB6 DB5 DB4 DB3 DB2 DB1 R5 R4 R3 R2 R1 R0 G5 G4 G3 G2 G1 G0 B5 B4 B3 B2 B1 B0

图3-213 TRI与DFM的配置

我们且看第一种配置TRI为0，DFM任意时的图形，低5位为蓝色B，中间6位为绿色G，最高5位为红色R；图中的是默认16条数据线时，像素点三原色的分配状况，DB1~DB5为5根线为蓝色，DB6~DB8以及DB10~DB12六根线为绿色，DB13~DB17这五根线为红色。这样分配有D0和D9位是无效的，使得刚好使用完整的16位。

RGB比例为5：6：5是一个十分通用的颜色标准，5+6+5=16位，表示5根线表示红色，6根线表示蓝色，5根线表示蓝色；如黑色的编码为0x0000，白色的编码为0xffff，红色为0xf800。比如红色0xf800，化成RGB5：6：5的二进制就是11111 000000 00000，

刚好可以看到R是红色的5位数据值是11111，G是绿色的值是000000，B是蓝色00000，因为红色寄存器内容都是1，而其他两种颜色都为0，所以最终会显示出红色。

3.31.4 控制器命令分析

什么是控制器，这个控制器在哪里？这个控制器可以用肉眼看得到吗？

控制器是看不到的，这个控制器主要负责一方面是与CPU沟通，另外一方面是去负责刷液晶屏的图像，它是看不见的，被压在液晶屏的内部了，这里主要通过查看液晶屏的数据手册就可以获得控制器IC的相关资料，CPU实际上就是与控制器IC进行通信，从而达到控制TFT 液晶屏的目的。

我们这里用IL9320的命令做分析，其他像SSD1289，IL9341，IL9328等都是同样的道理；下面我们了解几个重要的指令描述，如下图3-214、3-215中内容，这都是液晶屏的参数

手册中。

No.	Registers Name	R/W	RS	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
IR	Index Register	W	0	-	-	-	-	-	-	-	-	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0
SR	Status Read	R	0	L7	L6	L5	L4	L3	L2	L1	L0	0	0	0	0	0	0	0	0
00h	Driver Code Read	R	1	1	0	0	1	0	0	1	0	0	0	1	0	0	0	1	0
00h	Start Oscillation	W	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	OSC
01h	Driver Output Control 1	W	1	0	0	0	0	0	SM	0	SS	0	0	0	0	0	0	0	0
02h	LCD Driving Control	W	1	0	0	0	0	0	1	B/C	EOR	0	0	0	0	0	0	0	0
03h	Entry Mode	W	1	TRI	DFM	0	BGR	0	0	HWM	0	ORG	0	I/D1	I/D0	AM	0	0	0
04h	Resize Control	W	1	0	0	0	0	0	0	RCV1	RCV0	0	0	RCH1	RCH0	0	0	RSZ1	RSZ0
07h	Display Control 1	W	1	0	0	PTDE1	PTDE0	0	0	0	BASEE	0	0	GON	DTE	CL	0	D1	D0
08h	Display Control 2	W	1	0	0	0	0	FP3	FP2	FP1	FP0	0	0	0	0	BP3	BP2	BP1	BP0
09h	Display Control 3	W	1	0	0	0	0	0	PTS2	PTS1	PTS0	0	0	PTG1	PTG0	ISC3	ISC2	ISC1	ISC0
0Ah	Display Control 4	W	1	0	0	0	0	0	0	0	0	0	0	0	0	FMARKOE	FMI2	FMI1	FMI0
0Ch	RGB Display Interface Control 1	W	1	ENC2	ENC1	ENC0	0	0	0	0	RM	0	0	DM1	DM0	0	0	RIM1	RIM0
0Dh	Frame Maker Position	W	1	0	0	0	0	0	0	0	FMP8	FMP7	FMP6	FMP5	FMP4	FMP3	FMP2	FMP1	FMP0
0Fh	RGB Display Interface Control 2	W	1	0	0	0	0	0	0	0	0	0	0	0	VSPL	HSPL	0	DPL	EPL
10h	Power Control 1	W	1	0	0	0	SAP	BT3	BT2	BT1	BT0	APE	AP2	AP1	AP0	0	DSTB	SLP	0
11h	Power Control 2	W	1	0	0	0	0	0	DC12	DC11	DC10	0	DC02	DC01	DC00	0	VC2	VC1	VC0
12h	Power Control 3	W	1	0	0	0	0	0	0	0	VCMR	0	0	0	PON	VRH3	VRH2	VRH1	VRH0
13h	Power Control 4	W	1	0	0	0	VDV4	VDV3	VDV2	VDV1	VDV0	0	0	0	0	0	0	0	0
20h	Horizontal GRAM Address Set	W	1	0	0	0	0	0	0	0	0	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
21h	Vertical GRAM Address Set	W	1	0	0	0	0	0	0	0	AD16	AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
22h	Write Data to GRAM	W	1	RAM write data (WD17-0) / read data (RD17-0) bits are transferred via different data bus lines according to the selected interfaces.															
29h	Power Control 7	W	1	0	0	0	0	0	0	0	0	0	0	0	VCM4	VCM3	VCM2	VCM1	VCM0
2Bh	Frame Rate and Color Control	W	1	16M_EN	Dither	0	0	0	0	0	0	EXT_R	0	FR_SEL1	FR_SEL0	0	0	0	0

图3-214 重要指令描述

1）00h指令，当为读操作时，读取控制器的型号；当为写操作时，打开/关闭OSC振荡器，当写操作设置OSC比特位为1时，开启内部振荡器；为0时，停止振荡器。然后至少等待10ms时钟稳定后，再继续其它功能的设置。我们在代码设计中就是通过指令00h读取LCD控制器的型号，从而针对具体型号的控制器进行初始化操作，以便于兼容各种不同系列的LCD控制器。

No.	Registers Name	R/W	RS	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	
IR	Index Register	W	0	-	-	-	-	-	-	-	-	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0	
SR	Status Read	R	0	L7	L6	L5	L4	L3	L2	L1	L0	0	0	0	0	0	0	0	0	
00h	Driver Code Read	R	1	1	0	0	1	0	0	1	0	0	0	1	0	0	0	1	0	读型号
00h	Start Oscillation	W	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	OSC	打开osc
01h	Driver Output Control 1	W	1	0	0	0	0	0	SM	0	SS	0	0	0	0	0	0	0	0	
02h	LCD Driving Control	W	1	0	0	0	0	0	1	B/C	EOR	0	0	0	0	0	0	0	0	
03h	Entry Mode	W	1	TRI	DFM	0	BGR	0	0	HWM	0	ORG	0	I/D1	I/D0	AM	0	0	0	
04h	Resize Control	W	1	0	0	0	0	0	0	RCV1	RCV0	0	0	RCH1	RCH0	0	0	RSZ1	RSZ0	

图3-215 读液晶屏ID型号

2）03h指令，入口模式命令，如下图3-216：

03h	Entry Mode	W	1	TRI	DFM	0	BGR	0	0	HWM	0	ORG	0	I/D1	I/D0	AM	0	0	0	入口控制
04h	Resize Control	W	1	0	0	0	0	0	0	RCV1	RCV0	0	0	RCH1	RCH0	0	0	RSZ1	RSZ0	
07h	Display Control 1	W	1	0	0	PTDE1	PTDE0	0	0	0	BASEE	0	0	GON	DTE	CL	0	D1	D0	显示控制
08h	Display Control 2	W	1	0	0	0	0	FP3	FP2	FP1	FP0	0	0	0	0	BP3	BP2	BP1	BP0	

图3-216入口模式命令

我们重点关注的是I/D0、I/D1、AM这3个位，因为这3个位控制了屏幕的显示方向。AM控制GRAM的更新方向：当AM=“0”，表示地址更新方向为垂直方向；当AM=“1”，表示地址更新方向为水平方向；I/D[1：0]：当更新一个显示数据时，控制地址计数器自动增1和减1。详细内容如下图3-217所示：

	I/D[1:0] = 00 Horizontal : decrement Vertical : decrement	I/D[1:0] = 01 Horizontal : increment Vertical : decrement	I/D[1:0] = 10 Horizontal : decrement Vertical : increment	I/D[1:0] = 11 Horizontal : increment Vertical : increment
AM = 0 Horizontal	E B	E B	B E	B E
AM = 1 Vertical	E B	E B	B E	B E

图3-217 GRAM显示方向设置图

通过这几个位的设置，我们就可以控制屏幕的显示方向了，这种方法虽然简单，但是不是很通用，比如不同的液晶，可能这里差别就比较大，不能完全通用，比如9341和9320就完全不通用，具体要参考液晶屏手册。

3）07h，显示控制命令。该命令 CL 位用来控制是 8 位彩色，还是 26 万色。为 0 时 26 万色，为 1 时八位色。D1、D0、BASEE 这三个位用来控制显示开关与否的。当全部设置为 1 的时候开启显示，全 0 是关闭，如下图 3-218 所示。我们一般通过该命令的设置来开启或关闭显示器，以降低功耗。

R/W	RS	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
W	1	0	0	PTDE1	PTDE0	0	0	BASEE	0	0	0	GON	DTE	CL	0	D1	D0

图3-218显示控制命令

D[1:0]：设置为“11”时，打开显示；设置为“00”时，关闭显示；而配合BASEE比特位，可以用来设置开启或是关闭显示器时，系统是挂起还是运行状态，以此设置在挂起状态，达到降低功耗的目的，如下表3-135所示。

表3-135置开启/关闭显示器

D1	D0	BASEE	源，输出	ILI9320 内部操作
0	0	0	GND	停止
0	1	1	GND	操作
1	0	0	无亮点显示	操作
1	1	0	无亮点显示	操作
1	1	1	图像显示	操作

而CL比特位，当CL为1时，选择8位彩色；当CL为“0”时，选择为262144彩色，如下表3-220：

表3-136 选择8位彩色

CL	颜色（Colors）
0	262,144
1	8

4）20h和21h指令如下表3-137：

表3-137：20h和21h指令

20h	横向 GRAM 地址设置	W	1	…	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
21h	垂直 GRAM 地址设置	W	1	…	AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8

分别为设置GRAM的行地址（X坐标）和列地址（Y坐标）。我们通过此两个指令的设置，指定需要写入的点，然后再设置颜色，便实现在指定点写入一个颜色的；20h用于设置列地址（X坐标，0~239），21h用于设置行地址（Y坐标，0~319）。当我们要在某个指定点写入一个颜色的时候，先通过这两个命令设置到改点，然后写入颜色值就可以了。

5）22h指令如表3-138：

表3-138　22h指令写数据到GRAM区域

22h	写数据到 GRAM 区域	W	1		RAM 数据写(WD17-0)或读(RD17-0)是通过不同的数据总线传送的，根据需要来设置

写数据到GRAM命令，当写入了这个命令之后，地址计数器才会自动的增加和减少。该命令是我们要介绍的这一组命令里面唯一的单个操作的命令，只需要写入该值就可以了，其他的都是要先写入命令编号，然后写入操作数。

6）R50~R53，行列GRAM地址位置设置，如下图3-223。这几个命令用于设定你显示区域的大小，我们整个屏的大小为240*320，但是有时候我们只需要在其中的一部分区域写入数据，如果用先写坐标，后写数据这样的方式来实现，则速度大打折扣。此时我们就可以通过这几个命令，在其中开辟一个区域，然后不停的丢数据，这样就不需要频繁的写地址了，大大提高了刷新的速度。

表3-139　行列GRAM地址位置设置

	R/W	RS	**D15**	**D14**	**D13**	**D12**	**D11**	**D10**	**D9**	**D8**
R50h	W	1	0	0	0	0	0	0	0	0
R51h	W	1	0	0	0	0	0	0	0	0
R52h	W	1	0	0	0	0	0	0	0	VSA8
R53h	W	1	0	0	0	0	0	0	0	VEA8
			D7	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
R50h			HSA7	HSA6	HSA5	HSA4	HSA3	HSA2	HSA1	HSA0
R51h			HEA7	HEA6	HEA5	HEA4	HEA3	HEA2	HEA1	HEA0
R52h			HSA7	HSA6	HSA5	HSA4	HSA3	HSA2	HSA1	HSA0
R53h			HEA7	HEA6	HEA5	HEA4	HEA3	HEA2	HEA1	HEA0

HSA[7:0]/HEA[7:0]：指定区域的垂直方向上的起点和终点。通过设置HAS和HEA比特，以限制GRAM区域的垂直方向上的大小。

VSA[8:0]/VEA[8:0]：指定区域的水平方向上的起点和终点。通过设置VSA和VEA比特，以显示GRAM区域的水平方向上的大小。

如下图3-219所示：

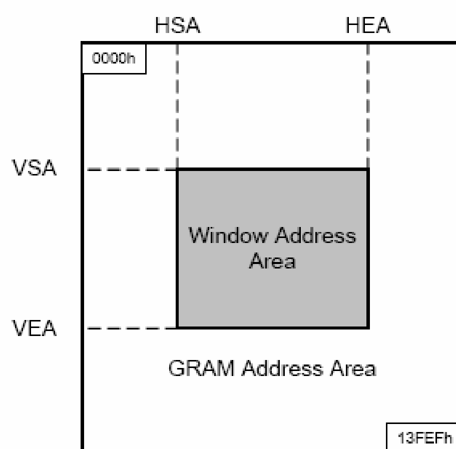


图3-219 位置的设置

其中：

“00”h ≤ HAS[7:0] ≤ HEA[7:0] ≤ “EF”h

“00”h ≤ VAS[7:0] ≤ VEA[7:0] ≤ “13F”h

可以看到 IL9320 中设置这个显示区域的代码如下，这里是设置 240-*320 大小：

```
LCD_WriteReg(0x50,0);    //Set X Start.
LCD_WriteReg(0x51,239);  //Set X End.
LCD_WriteReg(0x52,0);    //Set X Start.
LCD_WriteReg(0x53,319);  //Set X End.
```

命令部分到此，我们先简单了解到这里。有兴趣的朋友可以参阅《ILI9320控制器资料》一文，我们接下来看看要如何才能驱动TFTLCD模块，
TFTLCD显示需要的相关设置步骤如下：

1）初始化彩屏模块。

通过向彩屏模块写入一系列的设置，来启动TFTLCD的显示。为后续显示字符和数字做准备。比如：

【复位TFT】→ 【驱动IC初始化代码】→【复位所有的寄存器】→【开启显示】→【显存清0】→【开始显示】→【显示各种图画】

2）开始使用彩屏，通过彩屏模块的一些控制信号线来控制彩屏。
这里就是通过我们设计的程序，将要显示的字符送到彩屏模块就可以了，这些函数将在软件设计的时候介绍。通过以上两步，我们就可以使用 2.4 寸彩屏模块来显示字符和数字了， 并且可以显示各种颜色的背景。

3.31.5 控制器命令分析

本实验的硬件设计包括两个方面：

一、TFT LCD屏的原理设计，主要体现TFT LCD屏上的信号连接情况；

二、神舟51+ARM开发板上的2.4寸TFT座的信号连接情况。

液晶屏硬件接口电路图如下图3-220所示：

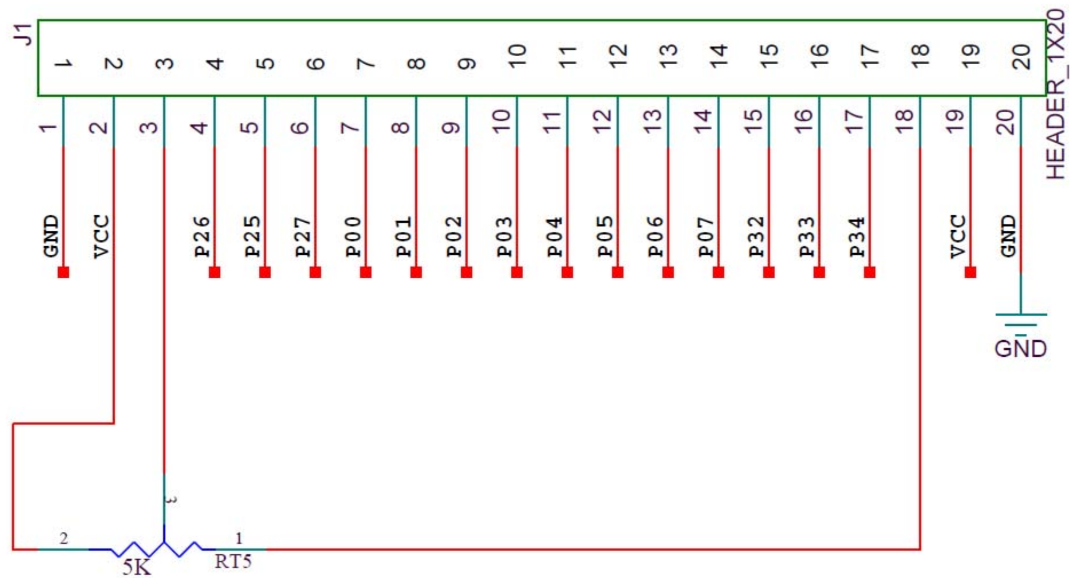


图3-220硬件电路图

首先是TFT的硬件设计如下图3-221：

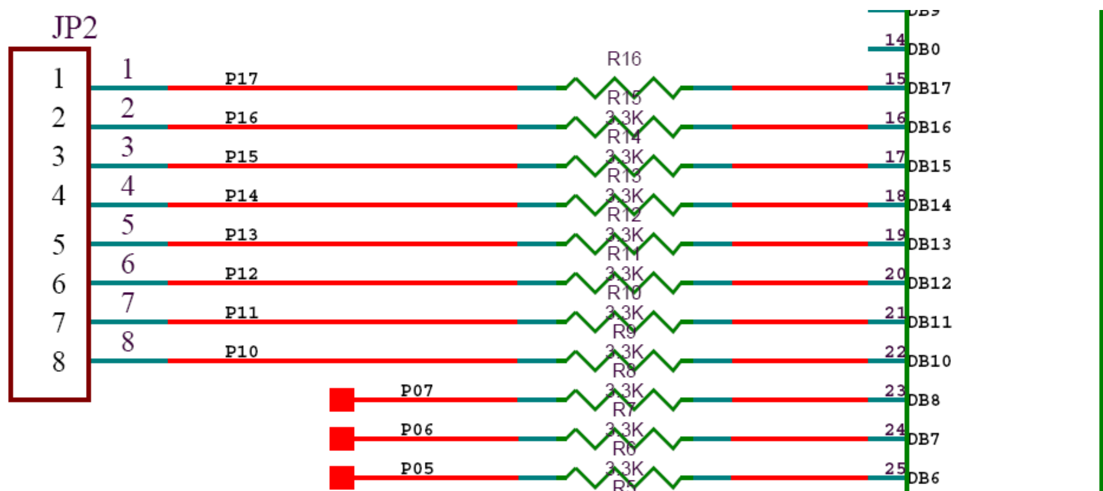


图3-221TFT的硬件设计

下表3-140是LCD的信号线的功能说明：

表3-140 LCD的信号线功能

LCD 信号线	对应管脚	功能
CSB	P2.7	TFTLCD 片选信号。

WRB	P2.5	向 TFTLCD 写入数据。
RDB	P3.2	从 TFTLCD 读取数据。
D[15:0]	P0 口、P1 口	16 位双向数据线。
RS	P2.6	命令/数据标志（0，读写命令；1，读写数据）

可以发现，对LCD屏的操作主要用到的管脚有CSB、WRB、RDB、RS、P0口8根数据线、P1口8根数据线，一共20根。通过将对应的信号线拉高拉低，数据通过P0口和P1口共16根数据线进行传输。

除了这些LCD使用的线之外，还有一些信号线比如SD_CS、TP_CS等等，这些我们没有用到，这些管脚我们用到的时候再给大家说明。

3.31.6 例程 01 TFT彩屏显示红色

代码如下：

```
#include <reg52.h>
#define   DataBusH      P1   //定义 P1 口 8 个管脚，数据总共是 16 位，这是高 8 位数据
#define   DataBusL      P0   //定义 P0 口 8 个管脚，数据总共是 16 位，这是低 8 位数据
sbit  CSB       =     P2^7;
sbit  RESETB =     P3^3;
sbit  RS       =      P2^6;
sbit  WRB      =     P2^5;
sbit      T_DCLK       = P2^4;    //SPI CLOCK
sbit      T_DIN        = P2^3;    //SPI DATA IN
sbit      T_DOUT       = P2^2;    //SPI DATA OUT
sbit      T_IRQ        = P2^1;    //IRQ
sbit  RDB       =     P3^2;
sbit  KEY =    P3^1;
sbit   T_CS          = P3^0;
void LLCD_WRITE_CMD(int cmd)
{
        RS = 0;        //命令/数据标志（1，读写数据；0，读写命令）
        RDB = 1;       //从 LCD 读取数据
        WRB = 1;     //向 LCD 写入数据
        CSB = 0;       //LCD 片选信号拉低，表示装载数据到数据线
        DataBusH = cmd>>8;         //写入高 8 位数据
        DataBusL = cmd;              //写入低 8 位数据
        WRB = 0;         //WRB 线需要置低，将要进行的是写入的操作
        ;                        //延时一下，让数据写进去
        WRB = 1;         //WRB 线置高，停止写操作
        CSB = 1;          //LCD 片选信号拉高，表示不装载数据
}
void LLCD_WRITE_DATA(int dataa)
{
```

```
        RS = 1;       //命令/数据标志（1，读写数据；0，读写命令）
        RDB = 1;   //从 LCD 读取数据
        WRB = 1;   //向 LCD 写入数据
        CSB = 0;   //LCD 片选信号选择 LCD 进行工作
        DataBusH = dataa>>8;   //写入高 8 位数据
        DataBusL = dataa;         //写入低 8 位数据
        WRB = 0;        //WRB 线置低，将写入的操作延时一下，让数据写进去
        WRB = 1;        //WRB 线置高，停止写操作
        CSB = 1;          //LCD 片选信号拉高，表示不装载数据
}
void send_host_reg_command(int cmd,int dataa)
{
       LLCD_WRITE_CMD(cmd);     //写命令到 TFT 液晶屏里
       LLCD_WRITE_DATA(dataa);   //写数据到 TFT 液晶屏里
}
void delayus(int value1)
{
      while (value1)
           value1--;
}
void delayms(int value)
{
      while (value)
      {
           delayus(99);
           value--;
      }
}
void LCD_INIT_ILI9328(void)
      {
//*********************************************Start intial Sequence
           send_host_reg_command(0x00e3,0x3008);
           send_host_reg_command(0x00e7,0x0012);
           send_host_reg_command(0x00ef,0x1231);
           send_host_reg_command(0x0001,0x0100);
           send_host_reg_command(0x0002,0x0700);
           send_host_reg_command(0x0003,0x1030);
           send_host_reg_command(0x0004,0x0000);
           send_host_reg_command(0x0008,0x0207);
           send_host_reg_command(0x0009,0x0000);
           send_host_reg_command(0x000a,0x0000);
           send_host_reg_command(0x000c,0x0000);
           send_host_reg_command(0x000d,0x0000);
           send_host_reg_command(0x000f,0x0000);
```

```
//*********************************************Power On
	send_host_reg_command(0x0010,0x0000);
	send_host_reg_command(0x0011,0x0007);
	send_host_reg_command(0x0012,0x0000);
	send_host_reg_command(0x0013,0x0000);
	delayms(200);
	send_host_reg_command(0x0010,0x1490);
	send_host_reg_command(0x0011,0x0227);
	delayms(50);
	send_host_reg_command(0x0012,0x001a);
	delayms(50);
	send_host_reg_command(0x0013,0x1400);
	send_host_reg_command(0x0029,0x0019);
	send_host_reg_command(0x002b,0x000c);
	delayms(50);
	send_host_reg_command(0x0020,0x0000);
	send_host_reg_command(0x0021,0x0000);
//*********************************************Set gamma
	send_host_reg_command(0x0030,0x0000);
	send_host_reg_command(0x0031,0x0607);
	send_host_reg_command(0x0032,0x0305);
	send_host_reg_command(0x0035,0x0000);
	send_host_reg_command(0x0036,0x1604);
	send_host_reg_command(0x0037,0x0204);
	send_host_reg_command(0x0038,0x0001);
	send_host_reg_command(0x0039,0x0707);
	send_host_reg_command(0x003c,0x0000);
	send_host_reg_command(0x003d,0x000f);
//**********************************************Set Gram aera
	send_host_reg_command(0x0050,0x0000);
	send_host_reg_command(0x0051,0x00ef);
	send_host_reg_command(0x0052,0x0000);
	send_host_reg_command(0x0053,0x013f);
	send_host_reg_command(0x0060,0xa700);
	send_host_reg_command(0x0061,0x0001);
	send_host_reg_command(0x006a,0x0000);
//*********************************************Paratial display
	send_host_reg_command(0x0080,0x0000);
	send_host_reg_command(0x0081,0x0000);
	send_host_reg_command(0x0082,0x0000);
	send_host_reg_command(0x0083,0x0000);
	send_host_reg_command(0x0084,0x0000);
	send_host_reg_command(0x0085,0x0000);
//********************************************* Plan Control
```

```
        send_host_reg_command(0x0090,0x0010);
        send_host_reg_command(0x0092,0x0600);
        send_host_reg_command(0x0007,0x0133);
}
//============================================================//
void LCD_WRITE_CMD(char cmd1,char cmd2)
{
    RS = 0;        //命令/数据标志（1，读写数据；0，读写命令）
    RDB = 1;       //从 LCD 读取数据
    WRB = 1;       //向 LCD 写入数据
    CSB = 0;       //LCD 片选信号选择 LCD 进行工作
    DataBusH = cmd1;    //写入高 8 位数据
    DataBusL = cmd2;    //写入低 8 位数据
    WRB = 0;            //WRB 线需要置低，以示即将要进行的是写入的操作
    ;                   //延时一下，让数据写进去
    WRB = 1;            //WRB 线置高，停止写操作
    CSB = 1;            //LCD 片选信号拉高，表示不装载数据
}
void LCD_WRITE_DATA(char dataa,char datab)
{
    RS = 1;        //命令/数据标志（1，读写数据；0，读写命令）
    RDB = 1;       //从 LCD 读取数据
    WRB = 1;       //向 LCD 写入数据
    CSB = 0;       //LCD 片选信号选择 LCD 进行工作
    DataBusH = dataa;   //写入高 8 位数据
    DataBusL = datab;   //写入低 8 位数据
    WRB = 0;            //WRB 线需要置低，以示即将要进行的是写入的操作
    ;                   //延时一下，让数据写进去
    WRB = 1;            //WRB 线置高，停止写操作
    CSB = 1;            //LCD 片选信号拉高，表示不装载数据
}
void LCD_TEST_SingleColor(char colorH,char colorL)
{
    /* 确定位置刷屏的位置，确定 GRAM 的区域，准备进行填充，在这里规划了
240*320 这么大区域  {   */
    int i,j;
    LCD_WRITE_CMD(0x00,0x50);          LCD_WRITE_DATA(0x00,0x00);
//水平地址开始的位置
    LCD_WRITE_CMD(0x00,0x51);          LCD_WRITE_DATA(0x00,0xef);
//水平地址结束的位置
    LCD_WRITE_CMD(0x00,0x52);          LCD_WRITE_DATA(0x00,0x00);
//垂直地址开始的位置
    LCD_WRITE_CMD(0x00,0x53);          LCD_WRITE_DATA(0x01,0x3f);
//垂直地址结束的位置
```

```
        LCD_WRITE_CMD(0x00,0x20);                   LCD_WRITE_DATA(0x00,0x00);
//水平 GRAM 地址设置
        LCD_WRITE_CMD(0x00,0x21);                   LCD_WRITE_DATA(0x00,0x00);
//垂直 GRAM 地址设置
        LCD_WRITE_CMD(0x00,0x22);   //Write Data to GRAM
    /* 确定位置刷屏的位置，确定 GRAM 的区域，准备进行填充，在这里规划了
240*320 这么大区域 }   */
    /* 填充颜色，这个屏包括的区域是有 240*320 个点  {  */
        for (i=0;i<320;i++)
        {
            for (j=0;j<240;j++)
            {
                LCD_WRITE_DATA(colorH,colorL);
            }
        }
    /* 填充颜色，这个屏包括的区域是有 240*320 个点  {  */
}
void main()
{
    LCD_INIT_ILI9328();
    while(1)
    {
        LCD_TEST_SingleColor(0xf8,0x00);    //display red
    }
}
```

液晶屏实验硬件连接关系如表 3-141

表 3-141 硬件连接关系

<table>
<tr><th>单片机接口</th><th>插座 1</th><th>方式</th><th>液晶屏接口</th><th>线缆</th><th>功能</th></tr>
<tr><td>P1 口</td><td>JP13</td><td>直连</td><td>P10~P17 管脚</td><td>1 根 8 针扁平电缆</td><td>控制液晶屏</td></tr>
<tr><td>单片机插座</td><td>J1</td><td>直连</td><td>20 根插座管脚</td><td>直接插入</td><td>控制液晶屏</td></tr>
<tr><td colspan="6">实验现象：下载程序后，按一下单片机开关，可以看到液晶屏从上到下，刷成红色，显示红色</td></tr>
</table>

硬件连接实物图如下图 3-222 所示：

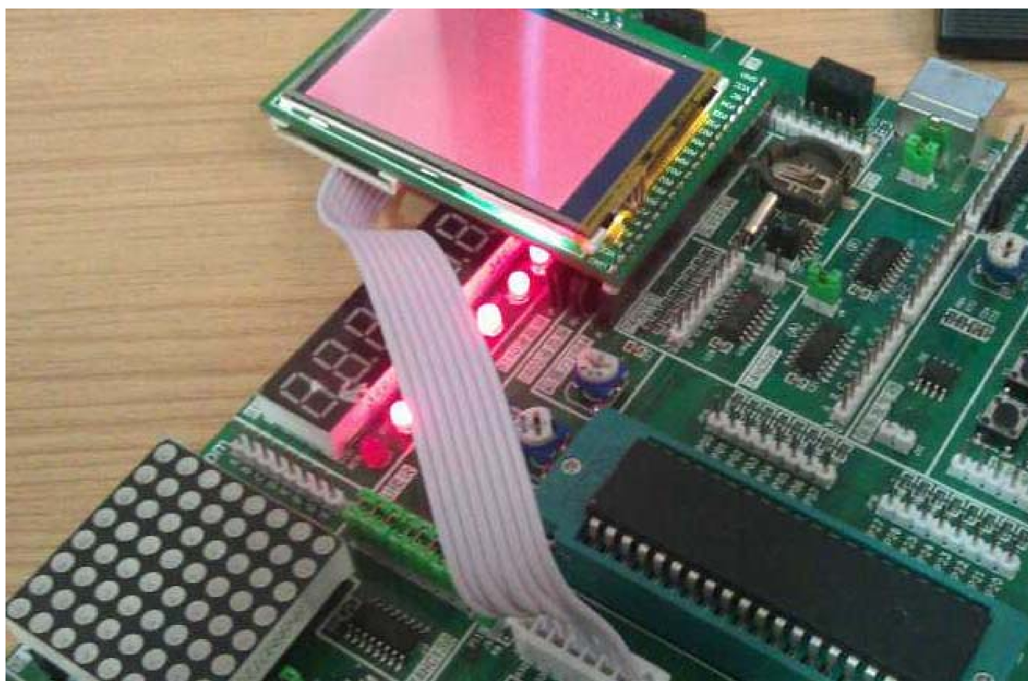


图3-222 硬件连接实物图

知识要点：

1. 该语句“void ISR_Key(void) interrupt 2 using 1”设置中断 2，并且在这个函数里对 P2 所连的 8 个 LED 灯进行取反。

2. 函数 LLCD_WRITE_CMD（）表示写命令到 TFT 液晶屏里；LLCD_WRITE_DATA()函数表示写数据到 TFT 液晶屏里，数据的传输通过单片机的 P0、P1 口传送，而液晶屏的控制则由 P2 进行控制具体请看下面的注释、代码与表 3-131：

```
#define DataBusH     P1 //定义 P1 口 8 个管脚，数据总共是 16 位，这是高 8 位数据
#define DataBusL     P0 //定义 P0 口 8 个管脚，数据总共是 16 位，这是低 8 位数据
void LLCD_WRITE_CMD(int cmd)
{
      RS = 0;    //命令/数据标志（1，读写数据；0，读写命令）
      RDB = 1;   //从 LCD 读取数据
      WRB = 1;   //向 LCD 写入数据
      CSB = 0;   //LCD 片选信号拉低，表示装载数据到数据线
      DataBusH = cmd>>8;       //写入高 8 位数据
      DataBusL = cmd;          //写入低 8 位数据
      WRB = 0;       //WRB 线需要置低，将要进行的是写入的操作
      ;              //延时一下，让数据写进去
      WRB = 1;     //WRB 线置高，停止写操作
      CSB = 1;     //LCD 片选信号拉高，表示不装载数据
}
void LLCD_WRITE_DATA(int dataa)
{
      RS = 1;    //命令/数据标志（1，读写数据；0，读写命令）
      RDB = 1;   //从 LCD 读取数据
      WRB = 1;   //向 LCD 写入数据
```

```
        CSB = 0;  //LCD 片选信号选择 LCD 进行工作
        DataBusH = dataa>>8;  //写入高 8 位数据
        DataBusL = dataa;      //写入低 8 位数据
        WRB = 0;      //WRB 线需要置低，将要进行的是写入的操作
        ;             //延时一下，让数据写进去
        WRB = 1;      //WRB 线置高，停止写操作
        CSB = 1;     //LCD 片选信号拉高，表示不装载数据
    }
```

51 单片机 I/O 口控制液晶屏功能说明如下表 3-142 所示：

表 3-142 51 单片机 I/O 口控制液晶屏功能说明

管脚名	信号名	说明
P2.7	CSB	拉低表示 LCD 片选信号被拉低，此时装载数据到 LCD 中
P2.6	RS	命令/数据标志（1，读写数据；0，读写命令）
P2.5	WRB	拉低时表示向 LCD 写入数据
P3.2	RDB	从 LCD 读取数据
P0.0	DataBusL	液晶屏总共是 16 位数据线，这是低 8 位数据线中的一位
P0.1	DataBusL	液晶屏总共是 16 位数据线，这是低 8 位数据线中的一位
P0.2	DataBusL	液晶屏总共是 16 位数据线，这是低 8 位数据线中的一位
P0.3	DataBusL	液晶屏总共是 16 位数据线，这是低 8 位数据线中的一位
P0.4	DataBusL	液晶屏总共是 16 位数据线，这是低 8 位数据线中的一位
P0.5	DataBusL	液晶屏总共是 16 位数据线，这是低 8 位数据线中的一位
P0.6	DataBusL	液晶屏总共是 16 位数据线，这是低 8 位数据线中的一位
P0.7	DataBusL	液晶屏总共是 16 位数据线，这是低 8 位数据线中的一位
P1.0	DataBusH	液晶屏总共是 16 位数据线，这是高 8 位数据线中的一位
P1.1	DataBusH	液晶屏总共是 16 位数据线，这是高 8 位数据线中的一位
P1.2	DataBusH	液晶屏总共是 16 位数据线，这是高 8 位数据线中的一位
P1.3	DataBusH	液晶屏总共是 16 位数据线，这是高 8 位数据线中的一位
P1.4	DataBusH	液晶屏总共是 16 位数据线，这是高 8 位数据线中的一位
P1.5	DataBusH	液晶屏总共是 16 位数据线，这是高 8 位数据线中的一位
P1.6	DataBusH	液晶屏总共是 16 位数据线，这是高 8 位数据线中的一位
P1.7	DataBusH	液晶屏总共是 16 位数据线，这是高 8 位数据线中的一位

3. 函数 send_host_reg_command（）表示写命令，写数据

```
    void send_host_reg_command(int cmd,int dataa)
    {
        LLCD_WRITE_CMD(cmd);     //写命令到 TFT 液晶屏里
        LLCD_WRITE_DATA(dataa);  //写数据到 TFT 液晶屏里
    }
```

那么 cmd 装载的是命令，dataa 是数据，可以进入到 TFT 液晶屏数据手册中如图 3-223，我们这里查看的是 9328 型号的数据手册，104 页的手册看到第 50 页的地方：

函数里的 cmd 就是指令 0x00h；dataa 表示对指令具体的设置，有 16 位，用 D0~D15 数据信号表示，通过设置这 16 位数据来改变对应指令的值，从而命令 TFT 液晶屏进行相应的操作和设置。

240RGBx320 Resolution and 262K color ILI9328

7.2. Instruction Descriptions

No.	Registers Name	R/W	RS		D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
IR	Index Register	W	0		-	-	-	-	-	-	-	-	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0
00h	Driver Code Read	RO	1		1	0	0	1	0	0	1	1	0	0	1	0	1	0	0	0
01h	Driver Output Control 1	W	1		0	0	0	0	0	SM	0	SS	0	0	0	0	0	0	0	0

图3-223 TFT液晶屏数据手册

首先，我们看看指令00h，当为读操作时，读取控制器的型号；当为写操作时，打开/关闭OSC振荡器。我们在代码设计中就是通过指令00h读取控制器的型号，从而针对具体型号的控制器进行初始化操作，以便于兼容ILI93XX系列的控制器。

当写操作设置 OSC 比特位为 1 时，开启内部振荡器；为 0 时，停止振荡器。然后至少等待 10ms 时钟稳定后，再继续其它功能的设置。

4.函数 LCD_INIT_ILI9328（）是具体初始化 TFT 液晶屏的函数，初始化过程就是发一系列的命令和设置给 TFT 液晶屏，代码如下

```
void LCD_INIT_ILI9328(void)
    {
//*********************************************Start intial Sequence
        send_host_reg_command(0x00e3,0x3008);
        send_host_reg_command(0x00e7,0x0012);
        send_host_reg_command(0x00ef,0x1231);
        send_host_reg_command(0x0001,0x0100);
        send_host_reg_command(0x0002,0x0700);
        send_host_reg_command(0x0003,0x1030);
        send_host_reg_command(0x0004,0x0000);
        send_host_reg_command(0x0008,0x0207);
        send_host_reg_command(0x0009,0x0000);
        send_host_reg_command(0x000a,0x0000);
        send_host_reg_command(0x000c,0x0000);
        send_host_reg_command(0x000d,0x0000);
        send_host_reg_command(0x000f,0x0000);
//*********************************************Power On
        send_host_reg_command(0x0010,0x0000);
        send_host_reg_command(0x0011,0x0007);
        send_host_reg_command(0x0012,0x0000);
        send_host_reg_command(0x0013,0x0000);
        delayms(200);
        send_host_reg_command(0x0010,0x1490);
        send_host_reg_command(0x0011,0x0227);
        delayms(50);
        send_host_reg_command(0x0012,0x001a);
        delayms(50);
        send_host_reg_command(0x0013,0x1400);
        send_host_reg_command(0x0029,0x0019);
```

```
        send_host_reg_command(0x002b,0x000c);
        delayms(50);
        send_host_reg_command(0x0020,0x0000);
        send_host_reg_command(0x0021,0x0000);
//********************************************Set gamma
        send_host_reg_command(0x0030,0x0000);
        send_host_reg_command(0x0031,0x0607);
        send_host_reg_command(0x0032,0x0305);
        send_host_reg_command(0x0035,0x0000);
        send_host_reg_command(0x0036,0x1604);
        send_host_reg_command(0x0037,0x0204);
        send_host_reg_command(0x0038,0x0001);
        send_host_reg_command(0x0039,0x0707);
        send_host_reg_command(0x003c,0x0000);
        send_host_reg_command(0x003d,0x000f);
//*********************************************Set Gram aera
        send_host_reg_command(0x0050,0x0000);
        send_host_reg_command(0x0051,0x00ef);
        send_host_reg_command(0x0052,0x0000);
        send_host_reg_command(0x0053,0x013f);
        send_host_reg_command(0x0060,0xa700);
        send_host_reg_command(0x0061,0x0001);
        send_host_reg_command(0x006a,0x0000);
//********************************************Paratial display
        send_host_reg_command(0x0080,0x0000);
        send_host_reg_command(0x0081,0x0000);
        send_host_reg_command(0x0082,0x0000);
        send_host_reg_command(0x0083,0x0000);
        send_host_reg_command(0x0084,0x0000);
        send_host_reg_command(0x0085,0x0000);
//******************************************* Plan Control
        send_host_reg_command(0x0090,0x0010);
        send_host_reg_command(0x0092,0x0600);
        send_host_reg_command(0x0007,0x0133);
}
```

看到这些代码，再从 TFT 液晶屏的数据手册中截取一部分寄存器进行分析，如下图 3-224：

a-Si TFT LCD Single Chip Driver
240RGBx320 Resolution and 262K color **ILI9328**

7.2. Instruction Descriptions

No.	Registers Name	R/W	RS	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
IR	Index Register	W	0	-	-	-	-	-	-	-	-	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0
00h	Driver Code Read	RO	1	1	0	0	1	0	0	1	1	0	0	1	0	1	0	0	0
01h	Driver Output Control 1	W	1	0	0	0	0	0	SM	0	SS	0	0	0	0	0	0	0	0
02h	LCD Driving Control	W	1	0	0	0	0	0	0	BC0	EOR	0	0	0	0	0	0	0	0
03h	Entry Mode	W	1	TRI	DFM	0	BGR	0	0	0	0	ORG	0	I/D1	I/D0	AM	0	0	0
04h	Resize Control	W	1	0	0	0	0	0	0	RCV1	RCV0	0	0	RCH1	RCH0	0	0	RSZ1	RSZ0
07h	Display Control 1	W	1	0	0	PTDE1	PTDE0	0	0	0	BASEE	0	0	GON	DTE	CL	0	D1	D0
08h	Display Control 2	W	1	0	0	0	0	FP3	FP2	FP1	FP0	0	0	0	0	BP3	BP2	BP1	BP0
09h	Display Control 3	W	1	0	0	0	0	0	PTS2	PTS1	PTS0	0	0	PTG1	PTG0	ISC3	ISC2	ISC1	ISC0
0Ah	Display Control 4	W	1	0	0	0	0	0	0	0	0	0	0	0	0	FMARKOE	FMI2	FMI1	FMI0
0Ch	RGB Display Interface Control 1	W	1	0	ENC2	ENC1	ENC0	0	0	0	RM	0	0	DM1	DM0	0	0	RIM1	RIM0
0Dh	Frame Maker Position	W	1	0	0	0	0	0	0	0	FMP8	FMP7	FMP6	FMP5	FMP4	FMP3	FMP2	FMP1	FMP0
0Fh	RGB Display Interface Control 2	W	1	0	0	0	0	0	0	0	0	0	0	0	VSPL	HSPL	0	DPL	EPL
10h	Power Control 1	W	1	0	0	0	SAP	0	BT2	BT1	BT0	APE	AP2	AP1	AP0	0	0	SLP	STB
11h	Power Control 2	W	1	0	0	0	0	0	DC12	DC11	DC10	0	DC02	DC01	DC00	0	VC2	VC1	VC0
12h	Power Control 3	W	1	0	0	0	0	0	0	0	0	VCIRE	0	0	PON	VRH3	VRH2	VRH1	VRH0
13h	Power Control 4	W	1	0	0	0	VDV4	VDV3	VDV2	VDV1	VDV0	0	0	0	0	0	0	0	0
20h	Horizontal GRAM Address Set	W	1	0	0	0	0	0	0	0	0	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
21h	Vertical GRAM Address Set	W	1	0	0	0	0	0	0	0	AD16	AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
22h	Write Data to GRAM	W	1	RAM write data (WD17-0) / read data (RD17-0) bits are transferred via different data bus lines according to the selected interfaces.															
29h	Power Control 7	W	1	0	0	0	0	0	0	0	0	0	0	VCM5	VCM4	VCM3	VCM2	VCM1	VCM0
2Bh	Frame Rate and Color Control	W	1	0	0	0	0	0	0	0	0	0	0	0	0	FRS[3]	FRS[2]	FRS[1]	FRS[0]
30h	Gamma Control 1	W	1	0	0	0	0	0	KP1[2]	KP1[1]	KP1[0]	0	0	0	0	0	KP0[2]	KP0[1]	KP0[0]
31h	Gamma Control 2	W	1	0	0	0	0	0	KP3[2]	KP3[1]	KP3[0]	0	0	0	0	0	KP2[2]	KP2[1]	KP2[0]
32h	Gamma Control 3	W	1	0	0	0	0	0	KP5[2]	KP5[1]	KP5[0]	0	0	0	0	0	KP4[2]	KP4[1]	KP4[0]
35h	Gamma Control 4	W	1	0	0	0	0	0	RP1[2]	RP1[1]	RP1[0]	0	0	0	0	0	RP0[2]	RP0[1]	RP0[0]

图 3-224　TFT 液晶屏部分寄存器

第二个指令为03h，进入模式命令。在这个命令中，我们关注AM、I/D1、I/D0三个比特位。

AM控制GRAM的更新方向：当AM=“0”，表示地址更新方向为垂直方向；当AM=“1”，表示地址更新方向为水平方向；

I/D[1：0]：当更新一个显示数据时，控制地址计数器自动增1和减1。详细内容如下 图3-225所示：

	I/D[1:0] = 00 Horizontal : decrement Vertical : decrement	I/D[1:0] = 01 Horizontal : increment Vertical : decrement	I/D[1:0] = 10 Horizontal : decrement Vertical : increment	I/D[1:0] = 11 Horizontal : increment Vertical : increment
AM = 0 Horizontal	E B	E B	B E	B E
AM = 1 Vertical	E B	E B	B E	B E

图3-225 更新方向的设定

第三个指令为07h，显示控制命令。

表-243显示控制命令

R/W	RS	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
W	1	0	0	PTDE1	PTDE0	0	0	BASEE	0	0	0	GON	DTE	CL	0	D1	D0

D1	D0	BASEE	源，输出	ILI9320 内部操作
0	0	0	GND	停止
0	1	1	GND	操作
1	0	0	无亮点显示	操作

1	1	0	无亮点显示	操作
1	1	1	图像显示	操作

D[1:0]：设置为“11”时，打开显示；设置为“00”时，关闭显示；而配合BASEE比特位，可以用来设置开启或是关闭显示器时，系统是挂起还是运行状态，以此设置在挂起状态，达到降低功耗的目的。

而CL比特位，当CL为1时，选择8位彩色；当CL为“0”时，选择为262144彩色，如下图3-244所示：

表3-244 显示彩色设置

CL	颜色（Colors）
0	262,144
1	8

接着我们了解第四个和第五个指令为20h和21h，分别为设置GRAM的行地址（X坐标）和列地址（Y坐标）。我们通过此两个指令的设置，指定需要写入的点，然后再设置颜色，便实现在指定点写入一个颜色的。

第六个指令为22h，（读/写）数据（到/从）GRAM。当这个指令执行时，地址计数器自动增加和减少。

下面再看看50h～53h指令，垂直和水平RAM地址位置设置如下表3-133所示。

表3-145 垂直和水平RAM地址位置设置

	R/W	RS	**D15**	**D14**	**D13**	**D12**	**D11**	**D10**	**D9**	**D8**
R50h	W	1	0	0	0	0	0	0	0	0
R51h	W	1	0	0	0	0	0	0	0	0
R52h	W	1	0	0	0	0	0	0	0	VSA8
R53h	W	1	0	0	0	0	0	0	0	VEA8
			D7	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
R50h			HSA7	HSA6	HSA5	HSA4	HSA3	HSA2	HSA1	HSA0
R51h			HEA7	HEA6	HEA5	HEA4	HEA3	HEA2	HEA1	HEA0
R52h			HSA7	HSA6	HSA5	HSA4	HSA3	HSA2	HSA1	HSA0
R53h			HEA7	HEA6	HEA5	HEA4	HEA3	HEA2	HEA1	HEA0

HAS[7:0]/HEA[7:0]：指定区域的垂直方向上的起点和终点。通过设置HAS和HEA比特，以限制GRAM区域的垂直方向上的大小。

VSA[8:0]/VEA[8:0]：指定区域的水平方向上的起点和终点。通过设置VSA和VEA比特，以显示GRAM区域的水平方向上的大小。如下图3-226所示：

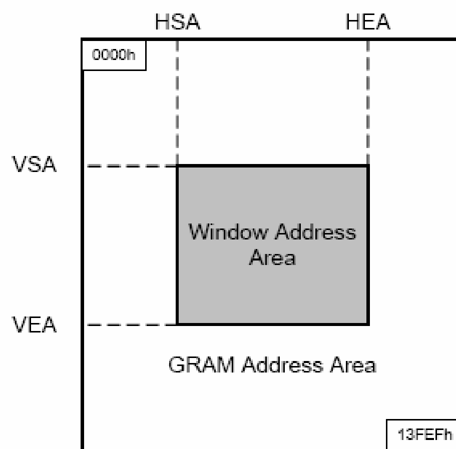


图3-226 指定区域设置

其中：

"00"h ≤ HAS[7:0] ≤ HEA[7:0] ≤ "EF"h

"00"h ≤ VAS[7:0] ≤ VEA[7:0] ≤ "13F"h

到此，我们先简单了解到这里，更多详细的请见数据手册。

5. LCD_TEST_SingleColor（）函数完成了颜色的填充，可以看一下代码，代码分为两个部分，第一部分完成颜色填充区域的设置和初始化；第二个部分完成屏幕颜色的实际填充，因为第一部分划分的区域包括了 240*320 个点，所以我们用一个 FOR 循环进行扫描来填充数据，具体请看下面的代码段：

```
void LCD_TEST_SingleColor(char colorH,char colorL)
{
	/* 确定位置刷屏的位置，确定 GRAM 的区域，准备进行填充，在这里规划了 240*320 这么大区域 {  */
	int i,j;
	LCD_WRITE_CMD(0x00,0x50);		LCD_WRITE_DATA(0x00,0x00);
//水平地址开始的位置
	LCD_WRITE_CMD(0x00,0x51);		LCD_WRITE_DATA(0x00,0xef);
//水平地址结束的位置
	LCD_WRITE_CMD(0x00,0x52);		LCD_WRITE_DATA(0x00,0x00);
//垂直地址开始的位置
	LCD_WRITE_CMD(0x00,0x53);		LCD_WRITE_DATA(0x01,0x3f);
//垂直地址结束的位置
	LCD_WRITE_CMD(0x00,0x20);		LCD_WRITE_DATA(0x00,0x00);
//水平 GRAM 地址设置
	LCD_WRITE_CMD(0x00,0x21);		LCD_WRITE_DATA(0x00,0x00);
//垂直 GRAM 地址设置
	LCD_WRITE_CMD(0x00,0x22);	//Write Data to GRAM
	/* 确定位置刷屏的位置，确定 GRAM 的区域，准备进行填充，在这里规划了
```

```
240*320 这么大区域 ｝  */

        /* 填充颜色，这个屏包括的区域是有 240*320 个点  ｛ */
         for (i=0;i<320;i++)
         {
             for (j=0;j<240;j++)
             {
                  LCD_WRITE_DATA(colorH,colorL);
             }
         }
        /* 填充颜色，这个屏包括的区域是有 240*320 个点  ｛ */
    }
```

上面有一句代码是：

LED_WRITE_CMD(0X00, 0X50); LCD_WRITE_DATA(0x00, 0x00); //水平地址开始的位置

可以在 TFT 液晶屏数据手册中找到对应的，如下：

表 3-146 水平地址开始的位置设置

50h	水平地址开始位置	……	HSA7	HSA6	HSA5	HSA4	HSA3	HSA2	HSA1	HSA0

这句代码就是设置 50H 这个位置，Horizontal Address Start Position 翻译成中文意思就是水平地址开始的位置，其他也是相同的，具体还是查手册吧。

6. 为什么会显示红色，液晶屏整个屏幕都显示红色，可以看这个代码

    LCD_TEST_SingleColor(0xf8,0x00); //display red

它将 0xf800 写入进去，因为在这个 16 位的液晶屏显示模式中，我们配置的 RGB 比例为 5：6：5 是一个十分通用的颜色标准，在 GRAM 相应的地址中填入该颜色的编码，即可控制 LCD 输出该颜色的像素点。刚好红色的编码就为 0xf800，RGB 比例为 5：6：5 总共为 16 个 bit，0xf800 化成二进制就是‘1111 1 000 0000 0000’， 将这个数据通过程序写到 LCD 中去刚好就可以得到如下图 3-227：

system 16-bit interface (1 transfers/pixel) 65,536 colors

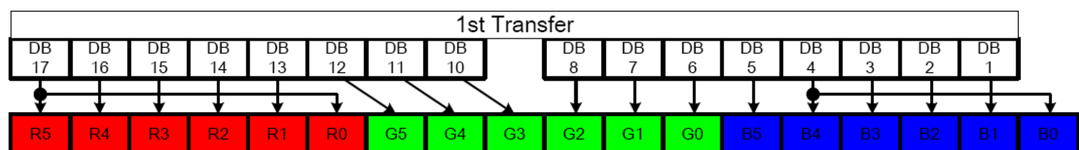


图 3-227 颜色设置

看上图，总共 18 位，但 R5 和 R0 的值一样；B4 和 B0 的值一样，去掉这 2 位，就是 16 位数，我们把 0xf800 化成二进制填写到下表 3-147 中：

表 3-147 18 位数据的配置

R5	R4	R3	R2	R1	R0	G5	G4	G3	G2	G1	G0	B5	B4	B3	B2	B1	B0
1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0

可以看到所有 R 开头的都是为 1，G 开头的和 B 开头的都为 0，所以液晶屏刷红色，这样就全部搞清楚了，下面再增加几个其他的例程，进一步再熟悉熟悉。

我们要实现的是全屏显示。对 240x320 像素的 2.4 寸彩屏来说，它有 240x320=76800 个像素点。本例程中，我们通过两个 for 循环来实现对 76800 个像素点的操作。函数

LCD_WRITE_DATA(colorH,colorL)写入颜色数据。可以发现函数 LCD_WRITE_DATA（）实际每次写入一个像素点的数据，一共写了 320x240 次，刚好写入显示一个屏幕所需的数据。

3.31.7 更多有关彩屏例程

更多 TFT 彩色液晶屏相应的例程可通过北京航空航天大学出版社下载专区下载，如下表 3-148：

表 3-148 TFT 彩色液晶屏更多丰富例程介绍

序号	例程功能
例程 01	2.4 寸显示红色彩屏
例程 02	2.4 寸显示绿色
例程 03	2.4 寸显示蛋黄色
例程 04	文字显示-欢迎进入神舟科技
例程 05	2.4 彩屏+SD 卡
例程 06	2.4 彩屏+SD 卡显示详细的 SD 卡信息

3.32 UCOSII操作系统的基础理解

3.32.1 操作系统是什么？

操作系统的作用究竟是什么呢？简单来说就是管理 CPU 的硬件资源的，例如一个 CPU 芯片既要负责网口数据的采集和分析，又要负责后台工作人员通过串口对 CPU 进行管理的配置和设置，还要负责把正在执行的一些数据信息显示到彩色液晶屏上，那么这样三件工作，在没有操作系统的时候，是需要采用 CPU 的中断来执行，或者我们自己写一个软件变量，来监控这 3 件事情，当事情发生的时候，就切换当前的事情不做，转去做着急的那件事情，或者在人所不能感知的条件下，CPU 进行分时间片来执行，每个事情都占用很少的一个时间，比如几个微妙或者几个毫秒等，然后轮流切换到其他事情。

可以看到以上这个问题如果单独由用户自己写代码来实现，会比较麻烦，而且可能很多代码来管理这些事情并不是最有效率的，所以为了管理好这么多事情，就引入了操作系统这个概念，操作系统专门负责管理多任务多事件的，可以非常合理的管理好硬件资源。

比如以前的 UCOSII 操作系统最多可以管理 64 个任务，但从 2.80 版本开始已经增加至 255 个任务，也就是说同时管理 255 个任务，多么强大的操作系统啊!

以下是μC/OS-Ⅱ使用的一些例子，因为 UCOSII 比较小，所以可以使用到比较低端的 CPU 上进行一些硬件资源的管理；因为如果较大的操作系统，操作系统本身就会消耗比较多的 CPU 资源，这样就很不划算；如果把 UCOSII 放到较高性能的 CPU 上，本身 UCOSII 比较简单，自己的功能还不足以去管理那么多高性能的 CPU。

3.32.2理解操作系统的小例子

关于操作系统的使用，举个小例子，程序都是从 main()函数开始，假设进入 main()函数

后，开始调用 A()函数，在 A 函数中发生某种状态的时候，就会调用 B()函数；如果没有发生这种状态，那么 A()函数将继续工作。这是没有操作系统的一个简单程序设计思路。

那么如果用操作系统呢？这个程序的设计理念就不一样了，首先从 main()函数进入，然后就会自动运行一个管理操作系统内部资源的小函数，我们这暂称之为 C()函数（这个在操作系统里叫做守护进程），然后创建一个 A()任务（其实它是一个函数），然后再创建一个 B()任务。三个任务 A，B，C 按时间片轮询工作，通俗的讲就是三个任务轮流来运行，每个任务工作假设是 20 毫秒时间。这样三个任务可以各自负责各自的工作，非操作系统时，需要 A()函数判断某个状态成立才运行 B()函数这样的工作，现在就不需要了，这个工作可以让 B()函数来判断，判断状态成立，就继续运行 B()函数后续的程序，反之则切换其他任务。

3.32.3 高端操作系统原理架构深入理解

层次 1：任务。2 个任务 A 和 B 以每隔 20ms 时间片轮询的方式运行，因为 A 任务和 B 任务都是程序员写的软件代码，所以有可能使得 A 和 B 同时对一个变量赋值，比如 A 刚对变量清 0 还没来得急使用时间片时间就到了，这时 A 休眠 B 开始运行，假如此时 B 对该变量赋了非零值后就休眠了，当 A 任务再次运行使用该变量时，因为 A 不知道 B 修改过，所以还认为该变量是 0，但实际上可能不是，所以就有可能造成 A 任务执行过程中出错；为了保证不出错，这就需要程序员自己事先规定好任务 A 和任务 B 划分各自的存储区域，不能跨界访问其他任务的存储区域。代表操作系统有 UCOSII。

层次 2：进程。在操作系统概念了中出现了进程这个名词，其实进程就是一个任务，它主要区别是进程分配出来自己独立的临时变量区域，其他进程不允许访问，这就为程序员写代码时省了不少功夫，无须考虑进程之间会相互跨界去影响对方，实现这些实际上是对操作系统内核代码重新设计了，从而从根本上解决了跨界出错的问题，因为增加了不少内存保护机制和分配机制，就使得操作系统内核代码增多了和复杂了。其实进程也是一个任务，它在操作系统中只是一个专业化术语，表示进程与进程之间是有保护机制存在的。

层次 3：界面。图形用户界面最早的故事来自于乔布斯认为微软抄袭了苹果，然后鼠标的出现结束了冷冰冰的命令行与计算机交互的时代。图形用户界面是什么？是一个进程吗？答案 YES，它确实是一个进程，它是一个在显示器上显示桌面样子的进程，桌面上所有的小图标都代表一个准备运行的程序，如果用鼠标双击某个图标，图形用户界面进程就会去激活该图标所对应的程序；所以图形用户界面就像是一个服务员，它是为用户服务的；例如 windows 操作系统，手机的 IOS 和 Andriod 操作系统。

层次 4：网络。在线 WORD，PPT，EXCEL 办公；微信的公众号，微信的小应用等都属于网络类别的服务应用。如果要说操作系统是管理 CPU 与外设资源之间的关系，网络级别的操作系统就是管理多个 CPU 和各自外设资源之间的关系。访问在线 WORD 时本机无须安装 WORD 程序也可以访问是因为通过 IP 地址去访问远程服务器时，远程服务器默认传送给浏览器客户端一套模板，在浏览器里显示就是 WORD 样式了，并且里面还有跟 WORD 一模一样的命令按钮，原来比如建立一个表格，是在本机保存，而现在则通过网络传送到服务器上在服务器上建立一个 WORD，数据都保存在远端；这样的好处就是本机无须装载程序也可以拥有 WORD 的功能，另外就是当一台服务器负载较重时，可以分配给其他服务器负载较轻的服务器，因为网络上传输的都是网络数据包，只需要把这些网络数据包转发给另外的空闲服务器让它们之间建立关联即可，这样就出现了分布式操作系统，负载均衡服务器

等等一系列的新软件功能系统，原本管理本地一个CPU资源变成管理网络上多个CPU资源，增加了管理设备的数量以及服务量，相当于原本是一个小操作系统干的活，现在小操作系统通过网络交给网络上的大操作系统来干，大操作系统干完活就把结果交给小操作系统；所以是的用户可以更少的安装程序，CPU 性能也可以不用那么高，只要有网络，再大再重的任务都可以交给网络上的服务器群去解决，由大操作系统或者叫分布式操作系统来解决，从而实现节省资源的目的。

简单总结一下，操作系统存在的意义就是管理和节约硬件资源，CPU 运行的操作系统假如被称为一个节点，那么这个被运行操作系统就是管理这个节点的硬件资源，使得这个节点工作效率最大化，节点与节点之间就是操作系统与操作系统之间的通信，这个通信在互联网里是遵循 TCP/IP 协议的，而多个操作系统结合 TCP/IP 协议又组成了一个更加庞大的虚拟操作系统来负责管理哲学节点的资源，微信，百度云，阿里巴巴，淘宝网等等这些巨无霸哪个背后没有成千上万台服务器在默默无闻的工作呢？这些都是操作系统在管理资源，当你在访问淘宝网购物时，当你在微信上聊天时，当你在百度云盘上下载一部电影时，这个应用是在虚拟的操作系统上运行，经过一些策略调度到某台服务器，服务器本身自己也有管理本机资源的操作系统，经过一系列复杂的协议转换，最终找到你请求的资源并提供相应服务。

3.32.4操作系统的学习心法

操作系统无论是在软件领域或计算机领域都可以算得上一门非常之难的学问，如何管理资源，无数个夜晚，我看着拿几行经典到无法再简化的代码，品种各种各样的情况下这小小的几行代码所应对的复杂情况，真是经典之作，真是艺术的感觉！通过这几行代码可以品位到设计者的智慧，就好比是一个象棋残局，无论谁先走，这个设计者都能给出解决办法，而残局就摆在你面前，你却还是下不过设计者。操作系统内核几乎都是由这样的代码组成的，看上去非常简单一个一个的函数，里面的全局变量设计，指针，引用，内存分配等就好像是一个精密的手表，环环相扣，逻辑严密，品它的味道得看自己的功力有多深。

在我工作这些年来见过大牛，几乎都是操作系统水平到了一定境界，操作系统不是一门可以 7 天入门到精通的学问，它只能日积月累慢慢提高才可以，所以在行业内真正佩服的高手可能不是看他收入有多高，工资是多少，而可能真正的看水平，在 IT 或者电子行业高端领域中，虽然高手不一定都精通操作系统，但精通操作系统的一定是高手，他们可能是架构师，可能是研发部门经理，可能是技术经理，也有可能是学科带头人等；关于操作系统的学习实在是无法用一章全部讲完，这就是为什么本章写得比较少的原因，因为实在是无法用一章全部讲完讲透讲明白，本章节只能起到一个入门的作用，有了这套操作系统的思路架构就可以去应对即将到来的工作，然后再在工作中慢慢去理解和加深，更深的知识需要相关配套的经典教材另外学习。

学习分为两个方法，一个是从操作系统内核开始学起，另外一个是从简单的应用开始学习，逐渐深入；下面分两个方面分别谈谈我的看法：

从内核开始学起，对于时间充裕或者对操作系统比较感兴趣的学生，建议系统的从内核开始学起，从内核学习就是学思想，看操作系统是如何管理资源的，看如何管理突然并发的 100 个任务，让这些任务都能有效的执行，既不耽搁用户的实时感，又不耽搁打印机任务的执行；另外就是有限资源的管理，比如每个任务要占 8K 内存，总共只有 16K 内存，如何最优同时运行和管理 100 个这样的任务；请记住，只学管理思想，思维，当然基础好的同学也可以模拟操作系统代码里的数据结构，去思考一下为什么这么设计，可以编写简单小程序来验证这些架构，比如编一个属于自己的链表，头和尾相连，考虑各种情况看是否存在 BUG；假如只学习这些管理思想，终究有一天当你进入实战工作的时候，最终会有突然一点就通的

感觉，量变引起质变，所以大家一定会在工作中出现一种现象，有些重点大学毕业基础较好的毕业生刚开始第一份工作时会感觉动手能力较差，但经过两年左右时间很快成为公司骨干成员，提高非常快。

另外一种是从外向内学习，本书所例的 UCOSII 例程就是采用这种学习方式，即学即用逐渐深入，随着开发板的畅销，这种学习方式会越来越多，十多二十年前，因为工具很少价格昂贵，不得不重理论和轻实践，现在条件好了有条件使用开发板，电脑价格也非常便宜，互联网使得查找资料非常方便，本人觉得这种学习方式可能会更加的先进；知识知识，能用的知识才有意义，管理资源就是操作系统要干的事情，它的一切设计都是遵循这个原则，读懂看明白本书的程序，再根据各自基础不同，实际情况不同，可以用操作系统的思想角度去理解写的程序或者别人的程序，不要为了读书而读书，操作系统也只是一段代码，它只是一段管理资源的代码，当程序设计遇到不解时，带着问题去查看一下操作系统是如何去解决原理想通的问题的，给自己一些灵感，另外如果有其他的建议，可以给我发 EMAIL，把您的具体情况和问题发给我，我们可以一起共同探讨，一起提高和成长。

3.32.5UCOSII的任务及其状态

操作系统是多任务的，多个任务同时存在，但 CPU 同一时刻只能执行一个单任务，这样就要求其他任务处于其他非运行状态。

随着操作系统进一步优化，为了更好硬件资源，管理好任务，这样就对这些任务分了一些等级；就好像轻重缓急的对任务分了几种状态，比如睡眠态、就绪态、运行态、挂起态、被中断态等。

如果要知道真实的操作系统的设计理念，可以翻书来研究这几种态操作系统是如何定义和管理它们的，什么情况下来切换到什么态，最能优化管理硬件资源，体现出轻重缓急，真正能够智能的实现自动控制，下面分析一下几种状态的功能，有兴趣研究的可以仔细看看，不感兴趣的可以结合后面的实际例程再来返回过来看：

1）睡眠态

2）就绪态.

3）运行态

4）挂起态

5）被中断态

睡眠状态：睡眠态（DORMANT）指任务驻留在程序空间之中，还没有交给 ucos II 管理，把任务交给 ucos II 是通过调用下述两个函数之一：OSTaskCreate()或 OSTaskCreateExt()。当任务一旦建立，这个任务就进入就绪态准备运行。任务的建立可以是在多任务运行开始之前，也可以是动态地被一个运行着的任务建立。如果一个任务是被另一个任务建立的，而这个任务的优先级高于建立它的那个任务，则这个刚刚建立的任务将立即得到 CPU 的控制权。一个任务可以通过调用 OSTaskDel()函数返回到睡眠态，或通过调用该函数让另一个任务进入睡眠态。

等待状态：正在运行的任务可以通过调用两个函数之一将自身延迟一段时间，这两个函数是 OSTimeDly()或 OSTimeDlyHMSM()。这个任务于是进入等待状态，等待这段时间过去，下一个优先级最高的、并进入了就绪态的任务立刻被赋予了 CPU 的控制权。等待的时间过去以后，系统服务函数 OSTimeTick()使延迟了的任务进入就绪态。正在运行的任务期待某一事件的发生时也要等待，手段是调用以下 3 个函数之一：OSSemPend()，OSMboxPend()，或 OSQPend()。调用后任务进入了等待状态（WAITING）。当任务因等待事件被挂起（Pend），

下一个优先级最高的任务立即得到了 CPU 的控制权。当事件发生了，被挂起的任务进入就绪态。事件发生的报告可能来自另一个任务，也可能来自中断服务子程序。

3.32.6 UCOSII任务的控制块OS_TCB

任务的控制块 OS_TCB 是一个表，用来记录描述任务的一些属性。假如有多个任务，每个任务都一个等待工作的人，CPU 每个时刻都只能让一个人来工作，其他人都得排队。那么这个 OS_TCB 此时就相当于是一个身份证和工作进度表的综合信息的结合。UCOSII 操作系统最多可以有 64 个任务，就好比有 64 个人同时工作，每个人都有各自的事情干，每个人都是不同的个体

那么 CPU 只有 1 个，只能有 1 个人在工作，其他 63 个人都在下面排队，或者睡觉；但这 63 个人的工作有的是刚开始，有的只工作了一半的进度，有的可能已经快完成了，那么这些信息都记录在哪里呢？在操作系统里，这些任务的信息都记录在 OS_TCB 里。具体的细节可以参看一些相关的书籍，这里主要是说明原理。

3.32.7 UCOSII的就绪表

同样，这些任务中，有的任务还没有准备好，而有的任务已经准备好了；比如有个任务是要打印一些文档，那么这个时候打印机正在被另外一个任务占用，那么这个任务就叫还没准备好，只有等另外一个占用打印机的任务释放掉打印机，然后分配给了这个任务，那么这个任务就叫准备好了，那么就可以进入 UCOSII 就绪表。

3.32.8 UCOSII的任务调度

俗话说，远水救不了近火，凡事都有优先顺序，比如有个人普通的感冒，那么他可以去医院看普通的门诊，慢慢排队挂号，等待看医生；相反，如果有个人病到有生命危险，这个时候再去正常排队的话，可能会耽误最佳的救治时间，那么这个时候，他可以选择看急诊，急诊是 24 小时的，随去随看。可以看到医院都是分了优先级的，那么操作系统也是一样，所有任务也是有优先顺序级的，这是最科学的一种规定设置，所以 UCOSII 操作系统里的任务调度就是通过任务的优先级，合理管理任务，任务的调度确定哪个任务优先级最高，先运行哪个任务。

3.32.9 UCOSII的调度器上锁、开锁

这个属于对特殊情况的处理，比如某些任务，分给它单独占用 CPU 的时间已经用完，但任务还没完成，此时按照操作系统的规则，应该要把这个任务换下，但这个任务所负责的事情又是不能被打断的，这个时候该怎么办呢？比如说有根水管爆了，水管工人正在维修，维修到一半的时候，刚好水管工人到了下班时间，这个时候如果不继续维修，整个城市的居民今天都无法通上自来水，这个时候就给调度器上锁，告诉 CPU 说，水管工人必须在第一时间抢修好水管，否则后果会很严重。

给调度器上锁，可以禁止任务调度；给调度器开锁，允许任务调度，可以使得某个任务保持对 CPU 的控制权，不管是否有优先级更高的任务进入了就绪态，直到这个任务完成后

调用调度器开锁为止。

3.32.10 UCOSII的空闲任务

如果没有任何任务的时候，操作系统在干嘛呢？对，它实际上在运行一个空闲任务，因为 CPU 一直是在工作的，就好像你啥事都不干的时候，心脏还是跳动的一样。

操作系统会自动建立一个空闲任务，在没有其他任务进入就绪态时投入运行。它的优先级永远设为最低优先级。当有其他任务就绪之后，它的优先级都会比空闲任务高，它就会获得 CPU 控制权，运行这个就绪好的任务。

3.32.11 UCOSII中的中断

这里的中断是软中断，每个任务都有自己的优先级，它们通过优先级开始占用 CPU，占用 CPU 后又会有很多细节的处理，在任务内部需要中断，中断服务完成后又回到任务本身继续进行，也就是说，这个中断也是从属于该任务的一部分。

如果在中断过程中任务被切换，那么这个任务以及它自己的中断状态都会被压入堆栽，通俗说，任务下去排队，它所执行的过程状态以及数据都不会丢失，都会保存下来，留着它下次获得 CPU 控制权的时候，再运行。

3.32.12 UCOSII的时钟节拍

每个任务运行多久，占用多久的 CPU 控制权，就绪状态多久，这些都需要记录时间。

而 UCSOII 里这个叫做时钟节拍，操作系统要求用户提供一个周期性时钟源，来实现时间的延时和超时功能，时钟节拍应该在每秒钟发生 10~100 次，为了完成该任务，可以使用硬件时钟。

时钟节拍可以由定时器中断来完成，也可以放在任务中完成。

3.32.13 UCOSII的初始化

操作系统必须被初始化，以初始化所有的变量和数据结构。就好像一个小面馆，无论今天有没有客户来吃，你都要准备好面条，烧开的热水，准备好配菜原料，准备好一个干净的店面，以及桌子凳子啥的。

操作系统也是一样，初始化就会有许多的变量要准备，许多的数据结构表，要记录各个任务的状态，还要做好准备等待随时到来的意外情况。

3.32.14 UCOSII的启动

一切准备好之后，开始启动程序，从任务就绪表中找出哪个任务优先级最高的来运行。这里说得更加详细是这样的，假如我们这次运行 64 个任务，那么先初始化好 64 个任务并绑定好，准备好 64 个任务需要的数据，变量，表等，然后再一声令下，开始真正的启动。

这个 UCSOII 启动之前，无论你怎么创建任务也好，怎么样也好，这个操作系统都是没

有运行的，只有 UCOSII 启动，才叫真正的运行。

3.32.15 例程 01 UcosII单任务运行

代码如下：

```
/**************************************************
*名称：UCOSII 操作系统实验
*作者：www.armjishu.com
*版本：v1.0
*内容：信号量点灯
**************************************************/
#include <includes.h>
#define     TASK_STK_SIZE     30
OS_STK    xdata     Task1Stack[TASK_STK_SIZE];
void Delay(INT8U i)
{
    INT8U j,k;
    for(i; i > 0; i--)
        for(j = 0; j < 250; j++)
            for(k = 0; k <250; k++);
}
void    Task1(void)     //操作系统中运行的任务 1
{
    INT8U err;
    while(1)
    {
        P2 = 0xf0;   //点亮 P0 口 8 个管脚的低 4 个 LED 灯
        Delay(3000);
    }
}


void main (void)
{
    OSInit();       //* Initialize uC/OS-II, The Real-Time Kernel
    /*  建立任务 1(任务代码指针、传递参数指针、分配任务堆栈栈顶指针、任务优先
级) */
    OSTaskCreate( Task1,
                  (void *)0,
                  (void *)&Task1Stack[TASK_STK_SIZE-1],
                  2);
    OSStart();
}
```

UCOSII 实验硬件连接实物图如下图 3-228：

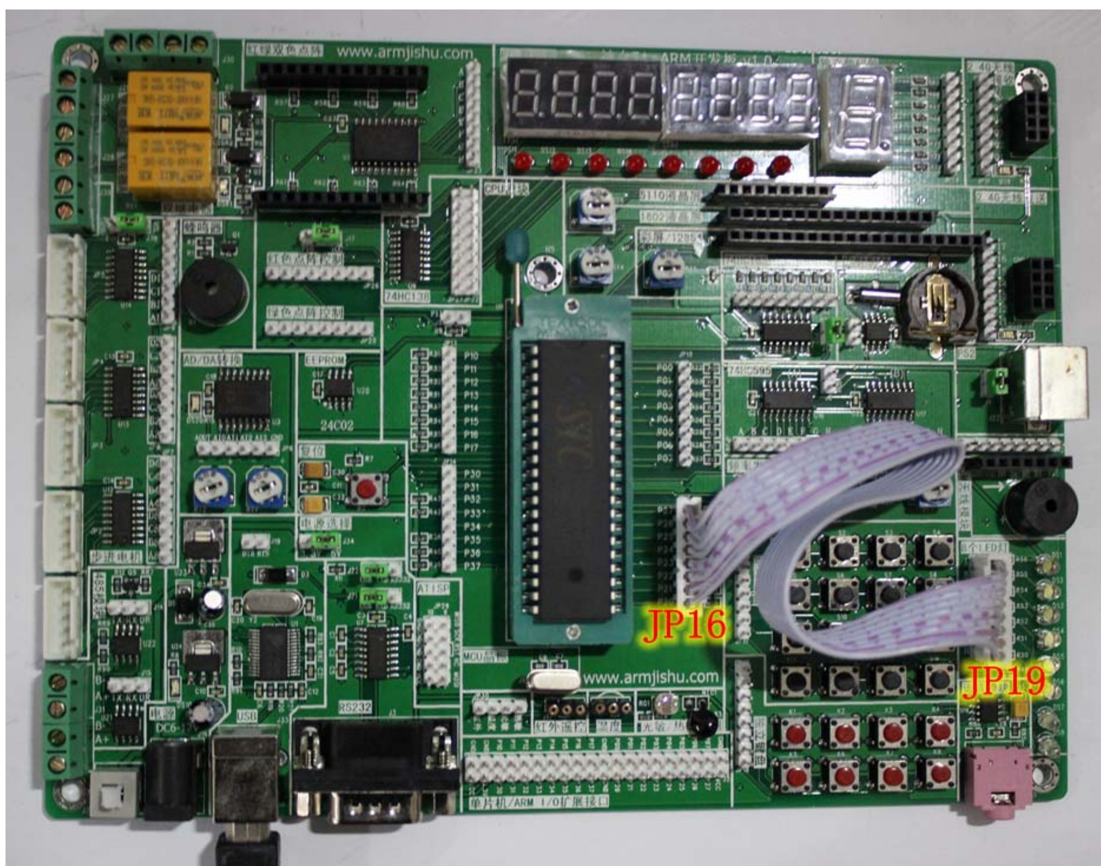


图 3-228 硬件连接实物图

UCOSII 实验硬件连接关系如下表 3-149 所示：

表 3-149 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P2 口	JP16(A 向左)	直连	JP19（A 向右）	1 根 8 针扁平电缆	控制 LED 灯
实验现象：JP19 端有 4 个 LED 灯持续被点亮					

知识要点：

1. OSInit（）初始化 UCOSII 操作系统

2. OSTaskCreate（）函数负责创建一个新的任务，这个函数里的参数 Task1 是一个单独的函数，这个任务一旦创建之后就会调用 Task1 这个函数，用户功能就可以在这个函数里进行定义

```
void   Task1(void)       //操作系统中运行的任务 1
{
     INT8U err;
     while(1)
     {
          P2 = 0xf0;       //点亮 P0 口 8 个管脚的低 4 个 LED 灯
          Delay(3000);
     }
}
```

3. OSStart()函数启动操作系统，运行任务

3.32.16 例程 02 UcosII多任务运行

代码如下：

```
/*************************************************
*名称：UCOSII 操作系统实验
*作者：www.armjishu.com
*版本：v1.0
*内容：信号量点灯
*************************************************/
#include <includes.h>
#define     TASK_STK_SIZE     30
OS_STK    xdata      Task1Stack[TASK_STK_SIZE];
OS_STK    xdata      Task2Stack[TASK_STK_SIZE];
OS_EVENT xdata * XinHaoLiangWait;
void Delay(INT8U i)
{
      INT8U j,k;
      for(i; i > 0; i--)
            for(j = 0; j < 250; j++)
                  for(k = 0; k <250; k++);
}
void   Task1(void)     //操作系统中运行的任务 1
{
      INT8U err;
      while(1)
      {
            P2 = 0xf0;
            Delay(3);
            OSSemPend(XinHaoLiangWait,10000,&err);    //等待一个信号量  OSSemPend
(OS_EVENT *pevent, INT16U timeout, INT8U *err)
      }
}
void   Task2(void )      //操作系统中运行的任务 2
{
      while(1)
      {
            P2 = 0x0f;
            Delay(3);
            OSSemPost(XinHaoLiangWait);     // 发送一个信号
      }
}
void main (void)
{
      OSInit();     //* Initialize uC/OS-II, The Real-Time Kernel
```

```
        XinHaoLiangWait = OSSemCreate(0);       //创建一个信号量
       /*  建立任务(任务代码指针、传递参数指针、分配任务堆栈栈顶指针、任务优先级) */
        OSTaskCreate( Task1,
                        (void *)0,
                        (void *)&Task1Stack[TASK_STK_SIZE-1],
                        2);
        OSTaskCreate( Task2,
                        (void *)0,
                        (void *)&Task2Stack[TASK_STK_SIZE-1],
                        3);
        OSStart();
    }
```

UCOSII 实验硬件连接实物图如下图 3-229：

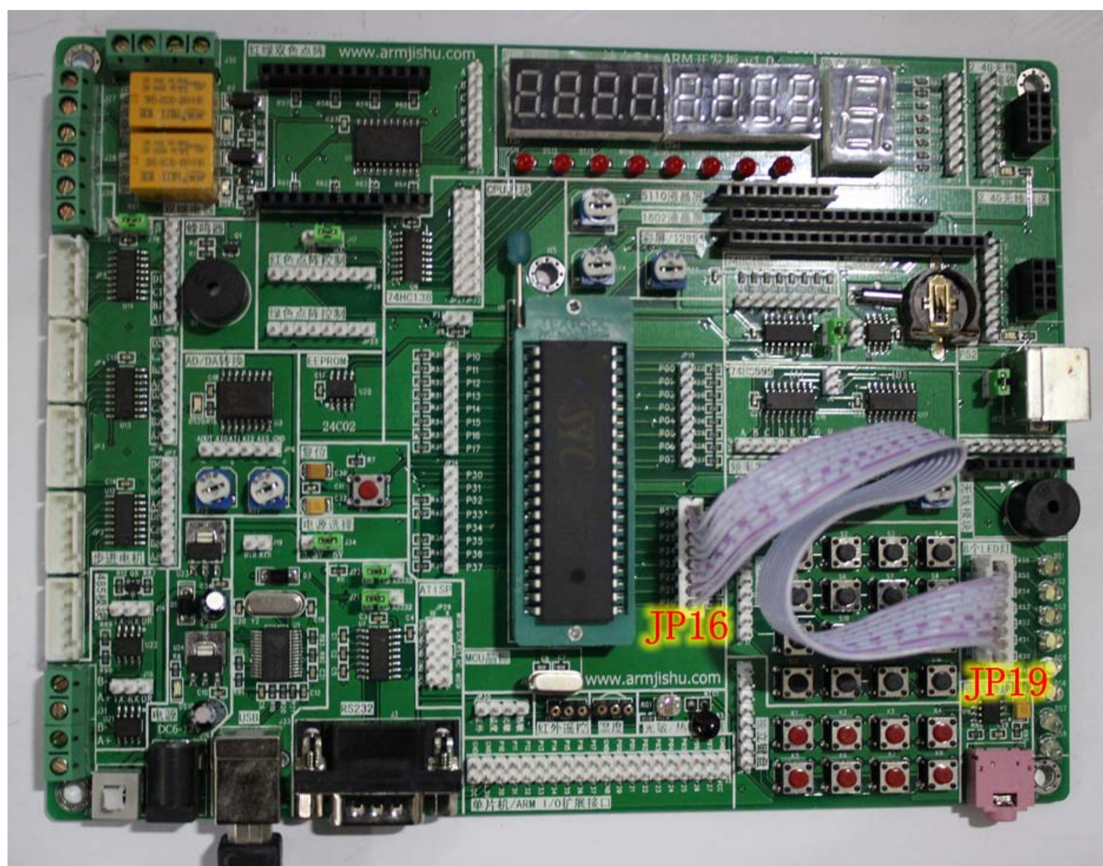


图 3-229 硬件连接实物图

UCOSII 实验硬件连接关系如下表 3-150 所示：

表 3-150 硬件连接关系

单片机接口	插座 1	方式	插座 2	线缆	功能
P2 口	JP16(A 向左)	直连	JP19（A 向右）	1 根 8 针扁平电缆	控制 LED 灯
实验现象：JP19 端有 4 个 LED 灯被点亮，另外 4 个是熄灭的，一闪一闪					

知识要点：

1. OSInit（）初始化 UCOSII 操作系统
2. XinHaoLiangWait = OSSemCreate(0)用来创建一个信号量
3. OSTaskCreate（）函数负责创建 2 个新的任务，建立任务(任务代码指针、传递参数指针、分配任务堆栈栈顶指针、任务优先级)， Task1 和 Task2 分别是 2 个单独的函数，各个任务一旦创建之后就会调用自己的这个用户功能定义函数：

```
void   Task1(void) //操作系统中运行的任务 1
{
    INT8U err;
    while(1)
    {
        P2 = 0xf0;
        Delay(3);
        OSSemPend(XinHaoLiangWait,10000,&err);  //等待一个信号量  OSSemPend
(OS_EVENT *pevent, INT16U timeout, INT8U *err)
    }
}
void   Task2(void )     //操作系统中运行的任务 2
{
    while(1)
    {
        P2 = 0x0f;
        Delay(3);
        OSSemPost(XinHaoLiangWait);   //  发送一个信号
    }
}
```

4.OSSemPost(XinHaoLiangWait);这个函数表示发送一个信号量，这个信号量发送给在等XinHaoLiangWait 变量的这个等待函数。
5.OSSemPend(XinHaoLiangWait,10000,&err)是表示等待一个名字叫 XinHaoLiangWait 的信号量，只有这个信号量到了之后，程序才会往下执行。
6.OSStart()函数启动操作系统，运行任务。

第四篇 ARM 理论基础深入篇

4.1 51 单片机与ARM处理器的区别

4.1.1 传统理念对 51 单片机和ARM的理解

传统观念中 51 单片机与 ARM 处理器有什么不同和区别呢，这里总结了大概

第一点，51 单片机速度相对 ARM 来说跑得慢，内部的 RAM 和 ROM 都比较小，单片机内部提供的接口不够丰富，比如串口、I2C，SPI，以太网等接口，单片机都跟 ARM 比起来，要

么是根本没有，要么有的话也是很有限。但是目前也出现了许多增强型的 51 单片机，也有不少丰富的接口集成进来了，比如带了 zigbee 的 51 单片机，比如集成了 USB 接口的 51 单片机等；但是因为主频速度慢，很多功能没法使用 51 单片机，比如使用 51 做以太网设备的公司应该是凤毛麟角了，未来是对网络速度要求越来越高，所以 51 单片机最终因为成本比较低，只能适用一些低端领域。

第二点，ARM 的主频比较高，速度很快，对许多工业应用，还有比如手持设备，包括手机等电子产品，速度越快越好，效率高，处理能力强，用户体验好。试问自己，使用高性能产品的感觉，越快越快好是永远不变的话题，就像 INTEL 的处理器，每隔一段时间就会推出速度更加快，价格更加便宜的处理器来一样。

第三点，ARM 内部 RAM 和 ROM 够多　使用 51 单片机时想定义一个稍微大一点的变量数组都不可能，因为 51 单片机的速度以及芯片内部寻址空间有限制(51 单片机一般是 8 位的)，所以限制了 RAM 和 ROM 的大小；51 也可以外挂 RAM 和 ROM，但 51 毕竟没有 ARM 的内部 RAM 和 ROM 够多。

第四点，ARM 处理器外围的资源非常的丰富，集成了很多接口，想 I2C、SPI、RTC、强悍的定时器、USB、以太网等等，甚至包括 ADC 和 DAC 等接口；但 51 单片机，如果需要使用 ADC 的时候，一般都会在 51 单片机附近外挂一个 ADC 的芯片，相当于单片机要与这颗 ADC 芯片进行通信才能完成这个功能，这样的话，成本也就增加了，原本 ARM 一颗芯片能够处理的事情，现在要用两颗芯片了；同样，例如 RTC 也是一样，如果 51 单片机方案要用 RTC 功能，必须外挂一颗 RTC 芯片才行，而 ARM 不需要。这样的好处，集成度高，不仅仅方便，而且还节约了电路板 PCB 的面积。

所以从这里可以看到，高端和低端是永远都会存在的两个方向，低端的产品成本低，高端的产品讲究品质，这两种需求就决定了 51 单片机虽然很慢，但是这么多年了，还是经久不衰，还没有被淘汰，目前很多领域都还是使用着 51 单片机。

4.1.2 51 单片机与ARM芯片内部的真正区别

20 世纪 80 年代到现在，单片机始终在 8 位机的档次上徘徊，8 位的单片机始终主导者应用的潮流。16 位单片机虽然也曾经掀起过波浪，但很快就销声匿迹了。随着科技的发展，人们对单片机的性能、速度、存储量、通信能力、功能的多样性，开发的方便程度及耗电的多少等不断提出更高的要求。32 位单片机应用的高潮正悄悄到来，ARM 处理器就是近年来十分迅猛的一种体系结构。

从 51 系列单片机到 ARM 处理器是一个很大的跨度，原因是 ARM 处理器完全不同于 51 系列单片机；一般称 51 系列单片机，单片机就是把中央处理器、存储器(RAM/ROM)和输入/输出设备集成在一个芯片内的芯片，是一个可以独立运行的最小智能系统。

而 ARM 不是单片机，而只是一个单片机的内核。单片机最主要的特征就是本身能组成最小系统，可独立运行，并具有完整的功能，而 ARM 则不能。ARM 和单片机不同，它仅仅是单片机中的中央处理器，一般称其为 ARM 处理器结构。也就是说以 ARM 为核，把 ARM 作为中央处理器，根据需要设计出外围功能模块，用总线把这些功能模块和 ARM 核连接在一起，组成一个单片机。这个单片机由 ARM 核控制，ARM 执行指令，并根据指令对外围设备发出各种控制命令。

那么 ARM 怎么样组成一个单片机呢？像 51 系列的中央处理器(内核)一样，ARM 不是单片机，是一个只能够做单片机的内核的中央处理器。它的任何功能实现都离不开外围设备，在这种 ARM 中央处理器的周围，可以设计出各种功能模块，比如 51 系列中所包含的串行通信模块、定时器/计数器模块等。由于这种模块设计相对独立，所以移植比较容易，由于

模块是通过总线进行连接的，模块的多少对 ARM 本身没有影响，因此，在一个片内围绕 ARM 中央处理器可以设计出很多不同的功能模块，所以以 ARM 为核非常容易和方便设计出功能强大的单片机。如下图 4-1 所示：

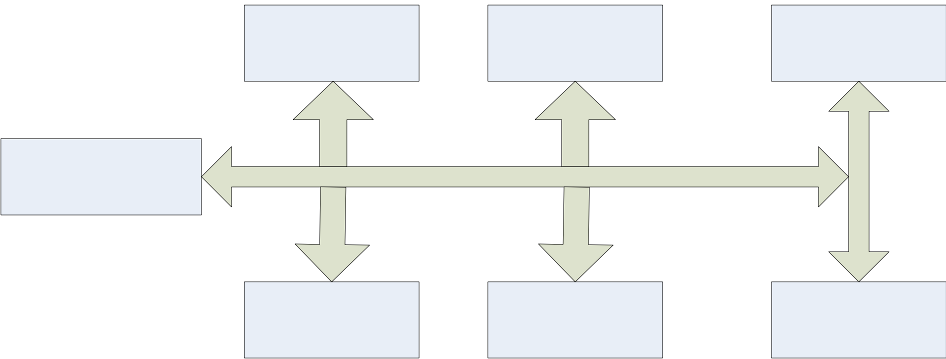


图 4-1 以 ARM 为核的单片机结构

在上图中，模块也可以是完全相同的或是几个相同的，这对系统并没有影响。同 51 系列相似，ARM 处理器对外部模块的操作仅仅是对不同地址的操作。每个模块都有自己的功能寄存器，例如可能有命令寄存器、状态寄存器和中断屏蔽寄存器等。每个模块的寄存器在总线上都有固定的地址，ARM 通过对这些地址的操作来实现对寄存器的控制和检测，也就是实现对这些模块控制和检测，从而实现这种功能；在这种设计中，ARM 核只是把外部模块作为不同的地址对象，各个模块的不同对 ARM 处理器来说，仅仅是地址不同而已。下表 4-1 为 51 系列和 ARM 处理器的比较。

表 4-1　51 系列和 ARM 处理器的比较

51 系列单片机内核	ARM7 处理器
8 位代码指令	32 位代码指令（兼容 16 位代码）
8 位数据总线	32 位数据总线
16 位地址总线	32 位地址总线
6 个中断源	7 个中断源（含复位）
工作寄存器（R0～R7）4 组	共 37 个寄存器
程序计数器	程序计数器
状态寄存器	状态寄存器
累加器 A 和 B	37 个都可以做累加器
寻址范围 16 位地址宽度	寻址范围 32 位地址宽度
不能预取指	三级流水线预取指
1 种工作模式	7 种工作模式
不支持协处理器	支持协处理器
不支持 JTAG 调试	支持 JTAG 调试

4.1.3 资深工程师谈谈芯片的性价比与如何芯片选型

我们在这里谈到 51 单片机到 ARM 的进阶之旅，就势必遇到一个问题，就是什么情况下使用 51 单片机，什么情况下要使用 ARM 处理器呢？并且 51 单片机和 ARM 芯片系列里面又被细分成许多种类，每个种类功能性能又都有差异，各自的侧重面不一样，比如有主要

侧重于玩具类的简单语音IC，比如有主要侧重于数字解码的DSP芯片等；也有需要5个串口的芯片，也有只有1个串口，1个USB接口，资源非常少的芯片；那么为什么有这么多不同呢？

主要原则是，性价比不一样，比如这颗ARM的芯片，如果ROM是256K要改变成512K的ROM，可能价格要增加1元钱；又比如同样一颗ARM芯片，要具备3路SPI的芯片与只有1路SPI接口的芯片，价格要贵2元；还有带以太网的ARM芯片与不带以太网的芯片可能价格相差3元；这里的相差价格都是为了举例而虚拟的，实际应用中，价格以各个厂商的实际报价为准，不过大体原理是这样的。

另外根据各自使用的用途也涉及到选型和成本，比如同样一个 100 管脚的芯片，比如STM32神舟II号开发板中的主芯片STM32F103VCT6与STM32神舟IV号开发板中的主芯片STM32F107VCT6都是100管教，而STM32F107VCT6支持以太网和OTG功能，这就使得这两个功能占用了不少管脚数，这两个功能都是STM32F103VCT6没有的，所以103VCT可以用一些其他功能来赋予这些管脚，这样，使得每颗各种型号不同的芯片都有各自侧重不同的用途，芯片从厂商生产出来后，管脚功能和数量都已经确定了的，所以这里也涉及到选型方面的考虑。

芯片选型可以说是硬件电子工程师必备的技能之一，一个好的芯片选型方案不仅可以节约产品成本、为客户省钱，而且可以节约硬件和软件工程师的开发时间、使产品尽早推向市场，从而使客户尽快享受电子信息时代给我们带来的便利。下面我们从几方面来总结一下芯片选型的要点：

芯片选型时需要关心的问题主要包括以下几个方面：性能、成本、功耗、芯片外围工作电路是否简洁方便、芯片的封装、货源是否充分、未来需求变化后方便升级和芯片寿命。

(1) 性能

之所以把性能放在第一位，是因为一个芯片的性能优劣决定了我们设计的产品的故障发生率及产品使用寿命，产品的故障发生率及使用寿命最终决定了我们公司的产品及品牌将在客户心中的地位，因为芯片选型时一定要保证所选的芯片在性能上不能有问题，至少在预见的时间范围内或在保修期范围内出现故障及损坏的几率很低。

芯片性能主要包括功能、工作温度、储藏温度、是否RoHS认证等，芯片选型时必须保证这几点符合产品的应用要求。例如产品是工业级的，那么芯片的工作温度一般必须在-40—+85 度范围之外。此外，有的产品还需要第三方认证，因此在性能的选取对于芯片选型至关重要。

很多人都知道一分价钱一分货的道理，而且很多客户在选择商品的时候往往并不会选择最便宜和最贵的，而且性价比比较高的，因此我们设计产品及选择何种芯片的最终目标其实也是为实现客户的所谓“最高性价比”而去努力。

(2) 成本

可以说，芯片的成本在一定程度上决定了产品的最终成本，因此芯片选型时必须考虑成本问题，很多时候我们都会发现，芯片贵一点，其功能和性能会好一点，而便宜的芯片往往在功能和性能上不及贵一点的，因此必须学会折中。那到底改怎么折中呢？我的建议是在功能满足需求的条件下以性能为主，要考虑产品未来可能会需要在性能上的提升。

(3) 功耗

现在的产品基本上都是标榜为“低成本、低功耗”，可以成本和功耗对于客户的重要性。芯片的功耗主要体现在芯片的工作电压和工作电流上，因此选型时可以通过芯片 datasheet中对工作电压和工作电流的指标描述可大致估计芯片工作时的功耗。目前的芯片很多都是宽

压输入及具有工作模式、睡眠模式、掉电模式等，因此可以考虑具有这样功能的芯片。

(4) 芯片外围工作电路是否简洁方便

芯片外围工作电路是指芯片的典型电路是否简洁，如果芯片只需接很少元件即能正常工作，那么选型时应该考虑使用这样的芯片，因此这时只需在芯片原有典型电路的基础上稍加上保护电路即可应用于产品设计中去。如果芯片典型电路的外围电路很多，那么不仅布局布线会很麻烦，而且也不利用产品的小型化设计和低成本设计理念；而且元件多还会影响生产加工的良品率，包括抗干扰等因素会增加。

(5) 芯片的封装

由于现在的产品都偏向于小型化设计，而芯片的封装在一定程度上决定了产品的最终尺寸，因此选择芯片时应考虑的封装问题。大的封装便于 PCB Layout、焊接及维修；但会加大板子尺寸从而增加产品成本，小的封装可以减小板子尺寸、降低产品成本但会加大 PCB Layout、焊接及维修的难度。选型时也应该需要考虑好这个问题。

(6) 货源是否充分

货源是否充分是指芯片选的是不是很冷门的芯片，如果选的芯片各方面都很不错，但是货源很少或者很难买到，那么这样的选型方案也是不可取的，首先你可能会因为这一颗料没买到或者供货周期太长而影响整个项目的进度，其次会为产品的售后维修带来很大的隐患。

(7) 未来需求变化后方便升级

一般资深的硬件工程师都会考虑这个问题，可能这个硬件方案可以满足目前的需求了，但老谋深算的硬件专家基本会考虑，如果需求改变了，要求更高了，改怎么办。所以他们会提前考虑好升级方案，比如当前需求 A 芯片已经足以满足，如果日后需求要改变，性能希望提高一倍，那么直接把 A 芯片换成 B 芯片即可，而且代码可以直接易植过来，改动非常小，而且不需要重新设计 PCB 电路板。因为 A 芯片和 B 芯片是 PIN 对 PIN 兼容的。

(8) 芯片寿命

芯片寿命是指芯片的寿命周期是从芯片厂商设计开始到样片推广，小批量，大批量生产，逐渐停产等的这个过程。一般来说，消费类电子的芯片寿命在四年-六年；工业级电子的芯片在 15 年以上。如果在选型的时候，没有关注到这一点，说不定经过艰苦 2，3 年研发产品终于稳定的时候，这个芯片就要停产了，马上又要考虑如何找替代方案，这样就会影响后续产品的生产和跟进，增加产品的成本。

4.2 从 51 到ARM的学习方法

4.2.1 精通 51 之后再来学习ARM

同学小 A 和小 C 两个人一起学习 51 和 ARM；小 A 这几个月来一直都爬在 51 单片机的学习上，他自己都有一点笑自己了，觉得自己比较笨，用了将近 4 个月的时间，专心巩固 51 的原理和程序，终于算是走过来了，对 51 有比较深入的了解了，小 A 说：“自己笨，身边的高才生又看不上 51 单片机，他们的水平都比较牛，说 51 太简单早已经过时了，要学就学 ARM，现在不是出了 CORTEX-A9 么？但是我不认为这是对的”，小 A 接着说：“51 是一

个基础，而且还很重要，如果 51 单片机学扎实了，再学 ARM 就事半功倍了，这是再我看了 ARM 之后感觉到的。它可以加速你的 ARM 学习速度，真得！不相信你试一试好了，真的是磨刀不误砍来工哦”。

小 C 的学习方法则是，51 单片机随便学学之后，又抓起 ARM 来学，但是因为 ARM 体系比较庞大，几个月内很难学完和消化，所以小 C 最终都是懂一点皮毛而已，51 和 ARM 都不精通，这样将导致在今后的工作中因为自身基础不够扎实，底子薄，同样学个新的东西或者完成一项任务，付出的努力要比别人大很多。

凡事都有一个客观规律，太过要求一个速度、效率，急功近利，可能不会得到一个好的结果，只有找准方法，稳打稳扎，拥有一本好的书就相当于拥有了一个好的老师，循序渐进，日日新，学一样就学精通一样，伤其十指不如断其一指。

4.2.2 市场上的ARM有哪些种类

目前关于 51 和 ARM 的发展，尤其是 ARM 的发展，可以用一片大好来形容，翻开各个公司的网站，招聘里面嵌入式方面的工作占据了大半页面，软硬件相互结合的工作用途是非常广的，而 ARM 又属于嵌入式领域的高端产品。

ARM 是微处理器行业的一家知名企业，设计了大量高性能、廉价、耗能低的处理器、相关技术及软件；ARM 公司本身并不靠自有的设计来制造或出售 CPU，而是将处理器架构授权给有兴趣的厂家；ARM 提供了多样的授权条款，包括售价与散播性等项目。对于授权方来说，ARM 提供了 ARM 内核的整合硬件叙述，包含完整的软件开发工具。许多半导体公司持有 ARM 授权：Atmel、Broadcom、Cirrus Logic、Freescale（于 2004 从摩托罗拉公司独立出来）、Qualcomm、富士通、英特尔（借由和 Digital 的控诉调停）、IBM，英飞凌科技，任天堂，恩智浦半导体（于 2006 年从飞利浦独立出来）、OKI 电气工业，三星电子，Sharp，STMicroelectronics 等。

说得更加通俗易懂一点，ARM 就是一个内核，一个处理器内核，各家公司购买这个内核之后，再自己把一些外设设计上去，比如加 2 个串口，加 1 个 JTAG 口，加 1 个 I2C 接口等设计出来的片子做成黑色的 IC 芯片，这样就成了一颗 ARM 处理器了；这些公司不需要设计 ARM 这个内核，就好像这些公司不用设计汽车的发动机一样，买来一个强劲的发动机，再自己组装成汽车（好比是这里的芯片处理器）。

4.2.3 ARM是硬件还是软件

ARM 到底是硬件还是软件，这里实际上说的 ARM 是硬件和软件结合，因为各个厂家设计的 ARM 外设接口都不相同，但是 ARM 架构都是一样，所以这颗 ARM 芯片的底层驱动就有相类似的地方，也有不同的地方，如果要使得这颗 ARM 芯片正常的运转起来，首先需要得到厂家的内部驱动程序（因为 ARM 芯片是他们生产的），然后再在这个驱动程序之上做许多的应用程序，应用程序需要比较多的软件工程师，比如有 Linux、Andriod、Wince、UCOSII 等等一些嵌入式的操作系统，这些操作系统之上还分有许多的软件方向，比如文件系统、GUI 界面、USB 接口、CAN 通信、485 通信、以太网 TCP/IP 协议以及驱动，而以太网中又分为很多细分软件分支，比如 FTP、HTTP 等；而这些软件工程师不太需要了解硬件知识，因为有驱动工程师把驱动这一层的软件接口做好即可。

所以到底是做驱动工程师还是应用工程师取决于对硬件的侧重还是对软件的侧重，但是

硬件工程师一定是要跟硬件接触的，因为各个厂家生产的 ARM 芯片都不同，所以同样都是 ARM11 的芯片，出来的芯片管脚定义可能都各不相同，那么设计出产品来，引出的电路板设计也有区别，这里需要硬件工程师仔细而认真的阅读芯片手册（芯片厂家提供）来设计硬件电路。

所以说，ARM 离不开硬件，也离不开软件，最牛的高手一般都是软硬件都精通的人，但是这样的人在中国真的是少之又少。

4.2.4 嵌入式主要的辅助调试工具有哪些

嵌入式开发需要辅助的调试工具目前来说，主要是 JLINK V8 仿真器，KEIL 公司出的 ULINK2 仿真器这两种最为常用。

那么 JLINK 与 ULINK2 有什么区别？相比两种仿真器，JLINK 用途更广，它可以支持 IAR，ADS，以及 KEIL 公司的 MDK 软件，它支持的 ARM 芯片内核有非常多的种类；而 ULINK2 是 KEIL 公司设计和生产的，它只支持 KEIL 公司的 MDK 软件，所以 ULINK2 的用途要比较窄一点。

那么刚才说到的两种仿真器跟编程器和下载器有什么区别？

编程器，Programmer，和程序员的英文一样。顾名思义，就是把已经编好的程序直接写入单片机（准确点应该是可编程器件）的机器，而不知道

仿真器，Emulator，是模拟的意思。我们可以用它来DEBUG，如断点，单步，全速等操作，可以知道软件运行过程中的状态。

下载器其实就是把程序下载到单片机里，这个功能编程器和仿真器都具备，具体的区别是编程器只负责下载程序到芯片里，假如有 1 万套产品要生产，一般都会先通过编程器批量烧录芯片（点一次烧录按钮，可以烧录多片芯片，有多个编程器烧录座可以插在编程器上，当然有的编程器也只有 1 个烧录座，不同型号的编程器不同）；而仿真器主要是调试和仿真功能，方便查看软件中的问题和 BUG。

如果学习 ARM，按照目前来说，基本上都建议购买一款仿真器，方便程序运行和在线下载，这几年仿真器都比较便宜了。

4.2.5 资深工程师眼中的嵌入式操作系统

资深嵌入式工程师眼中的操作系统有哪些呢？各有哪些特点

最小的操作系统非UCOSII莫属，搞开发和简单的工程设计非常好，因为它的小巧、多进程、简单，非常便于学习和理解操作系统的精髓，开发一些简单的任务系统比较合适。

UCLINUX属于比UCOSII稍微大一点的操作系统，同样它也是一个嵌入式的小型操作系统。

Linux是属于比较大的比较完善的嵌入式操作系统，目前最主流的操作系统，用途很广，像一些大的ARM嵌入式开发板都会必定支持Linux操作系统的，所有代码非常完整，并且全部开源，由开源社区来维护和维持这个系统。随着ARM芯片功能越来越强劲，速度越来越快，发挥Linux操作系统的功能就会越来越充分。

与Linux齐名的相类似功能的有微软公司出的Wince，还有Google公司出的Andirod的操作系统，比如现在许多手机都在使用Andirod嵌入式操作。

像51单片机只能跑跑UCOS操作系统或者UCLINUX，像Linux这样的操作系统，51单片机本身速度慢，内部的RAM比较小，根本都无法运行起来，像Linux这样的操作只能适合ARM处理器来运行。操作系统主要的功能是管理软硬件资源的，如果本身硬件资源不够多，不够

强劲，就不需要那么强大的智能的操作系统来对软硬件资源进行优化；因此，51单片机这样的最多10多M的主频，根本就不需要那么大的操作系统来管理的。

4.2.6 资深工程师眼中的嵌入式产品的开发流程

一．硬件设计(由硬件工程师负责)
1.产品硬件需求分析
2.产品硬件方案概要设计
3.产品硬件方案详细设计
4.设计原理图
5.设计PCB电路图
6.生产BOM元器件表和加工文件
7．制作PCB电路板
8.焊接元器件
9.测试和调试硬件
10.更新BUG重新修改设计文档，原理图，PCB，硬件完成

二．驱动设计（包含系统移植，由驱动工程师负责）
11.驱动各个接口，例如串口打印，LED点灯，按键，I2C，I2S等单个功能接口调试通过
12.操作系统易植（如果不需要操作系统这步则省略），操作系统与硬件底层有关系，所以在这里移植操作系统需要对驱动有所了解之后，才能做。
13.各个功能接口按照规范形成标准的接口函数，命名统一，方便调用
14.丰富各个功能接口函数支持各种不同的接口模式，最大效能发挥硬件优势，例如支持DMA方式以及普通方式，例如I2C支持I2C的通信方式也支持GPIO模拟I2C接口的方式等。

三．软件应用开发(由软件工程师负责)
15.产品功能需求分析，包括各项功能，功能主要是由软件来实现
16.产品软件功能概要设计，定义出软件框架和架构；软件的应用流程，逻辑流程等等，这里的内容非常丰富，这里只稍微提一下，以后再会详细讲解的。
17.产品软件功能详细设计，定义出全局变量，各个功能模块代码的具体实现，各个功能模块的共同区域，代码功能模块之间的信息如何进行交互，定义哪些 struct 结构体，宏定义，依据驱动工程师提供的函数，怎么进一步封装成高级应用函数。
18.代码实现
19.联机整体调试
20.代码修改和软件版本发布

4.2.7 ARM开发板的优点与缺点

ARM 开发板是一种学习板，一般主要有两种类别，一种是入门级的，第二种是入门之后想去学习的目标板。在传统的学习里，实际上就分为 51 单片机开发板和 ARM 开发板两种，因为嵌入式的芯片都是大同小异的。

芯片的控制其实最简单可以用三句话来描述：第 1 句话：芯片管脚不是输入，就是输出；第 2 句话：芯片管脚不是高电平，就是低电平；第 3 句话：管脚与管脚之间有一个传输协议，

例如 I2C、SPI、I2S 等。

无论是 51 单片机还是 ARM 芯片都无非是管脚之间的交互和控制，购买开发板主要是有以下几种用途：

1. 方便入门

有很多初学者，新出来一款芯片，大家都不熟悉，所以希望入门，这样选择入门级的开发板就要求这个开发板资料比较多，最好是比较详细，代码例程不要太过复杂，售后服务要仔细，细心；最好是有丰富的用户手册资料，可以从入门手把手的教您如何去学习。

2. 有可以参考的硬件

这款开发板有的硬件电路，就是新产品开发所希望用到的；例如 ENC28J60 的网口电路，新产品刚好需要，如果自己去看芯片手册，重新设计外围电路，那么还得自己设计代码来验证外围电路的正确性以及稳定性，如果刚好遇到一款稳定的开发板，有硬件电路的原理图，只要在这款开发板上验证硬件电路稳定就可以直接拿这款开发板的硬件原理图进行设计就足够了。

3. 有可以参考的软件

与有参考的硬件类似，有可以参考的软件代码也是对项目有很大帮助，例如有的开发板上已经移植好了强大的 GUI 界面或者不错以太网的代码，和文件系统代码等，就需要自己去移植了，只需要与目标板的电路一致，并且简单修改即可，还可以边设计硬件的时候，并行在开发板上先做一些软件开发的工作，所以在这个市场上，开发板的存在是非常有必要的。

神舟 51+ARM 这款开发板的定位主要是在方便入门上下了不少的功夫，当然也有一些可以参考的硬件，也有一些参考的软件，只是最大的优势还是在方便入门这一点上。

4.3 ARM编程入门

4.3.1 如何阅读STM32的芯片手册

首先介绍两个最重要的两个文档，芯片手册和参考手册。

1、芯片手册。芯片手册在网页“DATASHEET”那一栏。芯片手册详细介绍了所选择的芯片型号的功能规格，内核型号，运行主频，外设资源以及其性能，芯片封装种类信息以及各种封装的管脚定义，芯片的电气特性，外设的时序要求，订购信息和器件的机械特。这份文档因为芯片型号的不同而不同，比如 STM32F103C8 和 STM32F103ZE 和 STM32F105 以及 STM32F107 等等，他们的功能外设资源不同，所以芯片手册都不相同。在芯片的选型阶段，这份文档是判断芯片功能和性能是否满足项目需求的关键文档。在原理图阶段这份文档更是尤为重要。后续开发调试阶段这份文档也不可或缺。这份文档将伴随着你从项目开始一直走到项目结束。

2、参考手册。参考手册在网页“REFERENCE MANUALS”那一栏，也称为技术参考手册。这份参考手册是有关如何使用该产品的具体信息，包含各个功能模块的内部结构、所有可能的功能描述、各种工作模式的使用和寄存器配置等详细信息。参考手册不涉及某个具体的芯片，他是将一个系列的芯片。STM32F103C8 属于 STM32F10x 这一大类，所以我们下载的文档名为“RM0008: STM32F101xx, STM32F102xx、STM32F103xx、STM32F105xx 和 STM32F107xx，ARM 内核 32 位高性能微控制器”。也就是说无论你在 STM32F103C8 和 STM32F103ZE 和 STM32F105 以及 STM32F107 等芯片的网页页面下载的参考手册都是相同的。这也注定这份文档介绍的功能资源是 STM32F10x 这一大类芯片所有功能资源的交集。

这份文档包含了 USB 接口和以太网接口的介绍，但并不表示 STM32F103C8 包含这些接口。

这两份文档都很重要，相对来说，硬件开发人员更多关注芯片手册，软件开发人员更多关注参考手册。

其次介绍芯片相关的一些文档资料。

1、处理器内核相关文档。STM32F103C8 芯片的性能是 ARM 公司的 Cortex-M3。所以如果需要了解内核的资料，可以参考 ARM 公司的“Cortex™-M3 技术参考手册”以及其他 Cortex-M3 的技术书籍，例如“ARM Cotrex-M3 权威指南”等等。

2、ST 的应用笔记“APPLICATION NOTES”。充分利用 ST 网页中的资源，可以加快产品设计调试进度。

3、外设资源相关资料。例如 TIM 定时器、UART 串口以及 USB 等等。这些接口的资料可以参考 ST 网页中的 STM32 应用文档以及示例程序。由于很多接口都是各自的协议标准，所以可以查阅这些标准协议的相关资料文献，例如 I2C、SPI、USB 这些都有各自的规范可以查阅。

4、相关外部芯片资料。STM32 的接口对外连接了什么器件，就需要查阅相关的文档资料。同样是 I2C 接口，既可以连接 EEPROM 也可以连接温度传感器或其他；同样是 SPI 接口，既可以连接 DATA FLASH 也可以连接 WIFI 模块或者触摸屏等等。这些已经不属于 STM32 的范围了，所以这块资料在 STM32 的网页一般找不到，或者只能找到对应的参考设计。

5、开发板评估板手册。

6、论坛帖子。

4.3.2 STM32芯片的单个管脚是如何被控制的

1. 如何找到 STM32 芯片内部的寄存器地址

控制 STM32 芯片的单个管脚实际就是灵活控制管理 GPIO 管脚的芯片内部寄存器，那么我们怎么来访问这些寄存器呢？寄存器操作，首先要得到寄存器的地址。寄存器的地址是芯片厂商一开始就定好了的，固定的不能改变，大家看下图 4-2：

Address	Region
0x1FFF FFFF	reserved
0x1FFF F80F	Option Bytes
0x1FFF F800	System memory
0x1FFF F000	reserved
0x0801 FFFF	Flash memory
0x0800 0000	Aliased to Flash or system memory depending on BOOT pins
0x0000 0000	

Address	Region
0x4001 0400	AFIO
0x4001 0000	reserved
0x4000 7400	PWR
0x4000 7000	BKP
0x4000 6C00	reserved
0x4000 6800	bxCAN
0x4000 6400	shared 512 bit USB/CAN SRAM
0x4000 6000	USB Rregisters
0x4000 5C00	I2C2
0x4000 5800	I2C1
0x4000 5400	reserved
0x4000 4C00	USART3
0x4000 4800	USART2
0x4000 4400	reserved
0x4000 3C00	SPI2
0x4000 3800	reserved
0x4000 3400	IWDG
0x4000 3000	WWDG
0x4000 2C00	RTC
0x4000 2800	reserved
0x4000 0C00	TIM4
0x4000 0800	TIM3
0x4000 0400	TIM2
0x4000 0000	

Address	Region
0x4002 3400	CRC
0x4002 3000	reserved
0x4002 2400	Flash Interface
0x4002 2000	reserved
0x4002 1400	RCC
0x4002 1000	reserved
0x4002 0400	DMA
0x4002 0000	reserved
0x4001 3C00	USART1
0x4001 3800	reserved
0x4001 3400	SPI1
0x4001 3000	TIM1
0x4001 2C00	ADC2
0x4001 2800	ADC1
0x4001 2400	reserved
0x4001 1C00	Port E
0x4001 1800	Port D
0x4001 1400	Port C
0x4001 1000	Port B
0x4001 0C00	Port A
0x4001 0800	EXTI
0x4001 0400	AFIO
0x4001 0000	

图 4-2 ARM 寄存器地址

寄存器的地址是基址+偏移量的和。基址在芯片数据手册“存储器映像”章节，偏移量在芯片技术参考手册每一章节的最后部分。然后查阅技术参考手册获取每个寄存器是否可读写以及的每一比特信息，根据需要读写相关寄存器。

从芯片上电正常工作开始，所有寄存器都会有一个默认数值，保证处理器处于确定的状态。这个初始值可以在技术参考手册中获得。

寄存器的地址是基址+偏移量的和。基址在芯片数据手册“存储器映像”章节，正如上图那样，比如 Port A 的基址是’0x4001 0800’；比如 Port B 的基址是’0x4001 0c00’；比如 SPI1 的基址是’0x4001 3000’；比如 USART3 的基址是’0x4000 4800’；以此类推。

从芯片上电后工作正常开始，所有寄存器都会有一个默认数值，保证处理器处于确定的状态。这个初始值正如上图所描述的这样，这个重要内容我们可以在

技术参考手册中获得。

现在我们打开参考手册《STM32F10xxx 参考手册》STM32F103中文参考手册.pdf ，可以看到这是一个 524 页的参考手册，我们截图看看，如下图 4-3：



STM32F10xxx参考手册

参考手册

小，中和大容量的 STM32F101xx, STM32F102xx 和 STM32F103xx
ARM 内核 32 位高性能微控制器

导言

本参考手册针对应用开发，提供关于如何使用小容量、中容量和大容量的STM32F101xx、STM32F102xx或者STM32F103xx微控制器的存储器和外设的详细信息。在本参考手册中STM32F101xx、STM32F102xx和STM32F103xx被统称为STM32F10xxx。

STM32F10xxx系列拥有不同的存储器容量，封装和外设配置。

关于订货编号、电气和物理性能参数，请参考STM32F101xx、STM32F102xx和STM32F103xx的数据手册。

关于芯片内部闪存的编程，擦除和保护操作，请参考STM32F10xxx闪存编程手册。

图 4-3 STM32 参考手册

比如我们要访问 GPIO 管脚 PB8，那么首先就要找到端口 B 的位置，即 Port B,我们首先查看文档的目录，找到第 7.2 节 GPIO 寄存器描述章节，可以看到从 75 页开始到 77 页都是描述 GPIO 端口的，我们查看 7.2.1 节查看一下端口配置低寄存器（GPIOx_CRL），这个里的 x=A…E 表示，GPIOx_CRL 可以是 GPIOA_CRL、GPIOB_CRL、GPIOC_CRL、GPIOD_CRL、GPIOE_CRL 中的任意一个；在这里我只关心端口 B，就可以把它看成 GPIOB_CRL：如下图 4-4 所示

目录　　　　STM32F10xxx参考手册

图 4-4 GPIO 寄存器

我们可以找到 PortB 在内存中的基地址的 0x40001 0c00，控制端口 B 的各个寄存器的偏移地址应该怎么确定呢？可以查看具体的寄存器描述(PortB 属于 GPIO 端口中的 B 端口)，找到手册的”7.2.1 节　端口配置低寄存器（GPIOx_CRL）(x=A…E)”，可以看到该文档写了’偏移地址：0x00’，这表示 GPIOB_CRL 这个寄存器的地址等于 Port B 的位置+偏移地址就可以算出它的地址，即：

GPIOB_CRL 寄存器地址：0x4001 0c00（PortB 的地址）　+ 0x00（GPIOx_CRL 的偏移地址）= 0x4001 0c00

我们再看一个地址，在手册的7.2.2节 端口配置高寄存器GPIOx_CRH，它的偏移地址是0x04，复位值是0x4444 4444。

GPIOB_CRH 寄存器地址：0x40001 0c00（PortB 的地址） + 0x04（GPIOx_CRL 的偏移地址） = 0x4001 0c04。

也就是说，当我们访问 0x40001 0c04 这个地址所指向的内容时，实际上就是在访问 GPIOB_CRH 这个寄存器了，就这么简单。

2.知道寄存器地址后，如何查看和改变 STM32 处理器单个管脚的状态

这里主要是如何去查看 CPU 芯片单个管脚的寄存器表。

实际上，点亮这个 LED 灯，只需要使得我们的 STM32 芯片的 PB8 管脚输出低电平就可以了，那么如何控制一个 PB8 管脚的状态呢？

我们来举个例子，实际上 STM32 的 PB8 管脚的状态是由 STM32 芯片内部的一些寄存器来控制的，通过这些寄存器，可以控制将管脚配置成输出或者输入，拉高还是拉低。芯片通过获取寄存器不同的值，对应我们的 STM32 芯片手册寄存器说明书，就可以知道芯片就相当于获得了不同的命令，获取命令后就开始执行命令，大家可以看下图 4-5，还是看这个《STM32F103 中文参考手册》524 页的文档：



图 4-5 STM32 参考手册

可以看到文档的 7.2 节，注意文档目录红框里的这些寄存器，现在我们就开始仔细来研究一下它们，这是专门描述 GPIO 寄存器的章节，具体内容大家自己打开文档阅读一下，如下图 4-6：

目录 STM32F10xxx参考手册

图 4-6 GPIO 寄存器的章节

我们进入文档，浏览到第 75 页，如表 4-2、4-3 所示，我们看其中一个寄存器到底写了些什么，在手册的 7.2.1 节，端口配置低寄存器（GPIOx_CRL）（x=A…E），它的偏移地址是 0x00，复位值是 0x4444 4444：

表 4-2 端口配置低寄存器（GPIOx_CRL）

BIT 31	BIT 30	BIT 29	BIT 28	BIT 27	BIT 26	BIT 25	BIT 24
MODE15[1:0]		MODE14[1:0]		MODE131:0]		MODE12[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw

BIT 23	BIT 22	BIT 21	BIT 20	BIT 19	BIT 18	BIT 17	BIT 16
MODE11[1:0]		MODE10[1:0]		MODE9[1:0]		MODE8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw

BIT 15	BIT 14	BIT 13	BIT 12	BIT 11	BIT 10	BIT 9	BIT 8
MODE7[1:0]		MODE6[1:0]		MODE51:0]		MODE4[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT0
MODE3[1:0]		MODE2[1:0]		MODE1[1:0]		MODE0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw

表 4-3 端口配置低寄存器（GPIOx_CRL）功能说明

位31:30 27:26 23:22 19:18 15:14 11:10 7:6 3:2	**CNFy[1:0]**：端口x配置位(y = 0…7) 软件通过这些位配置相应的I/O端口，请参考表15端口位配置表。 在输入模式(MODE[1:0]=00)： 00：模拟输入模式 01：浮空输入模式(复位后的状态) 10：上拉/下拉输入模式 11：保留 在输出模式(MODE[1:0]>00)： 00：通用推挽输出模式 01：通用开漏输出模式 10：复用功能推挽输出模式 11：复用功能开漏输出模式
位29:28 25:24 21:20 17:16 13:12 9:8, 5:4 1:0	**MODEy[1:0]**：端口x的模式位(y = 0…7) 软件通过这些位配置相应的I/O端口，请参考表15端口位配置表。 00：输入模式(复位后的状态) 01：输出模式，最大速度10MHz 10：输出模式，最大速度2MHz 11：输出模式，最大速度 50MHz

我们看到如下内容，我们逐个列出来：

1） GPIOx_CRL 这个寄存器一共是 32 个比特，从 0-31bit，有 32 个位
2） GPIOx_CRL 寄存器的偏移地址是 0x00；表示在 GPIO 端口的基地址加上这个偏移地址 0x00 就可以访问到这个寄存器的内容。
3） GPIOx_CRL 寄存器的复位值是 0x4444 4444 总共 8 个‘4’
4） GPIOx_CRL 寄存器其中第 0、1 位和 4、5 位和 8、9 位和 12、13 位以此类推到 28、29 每两个位为一个组，叫做 MODEy 组；主要功能是设置这个管脚是输入模式，还是输出模式（如果是输出模式，还要确认输出速度是 10MHz、还是 2MHz、还是 50MHz）
5） GPIOx_CRL 寄存器其中第 2、3 位和 6、7 位和 10、11 位和 14、15 位以此类推到 30、31 每两个位为一个组，叫做 CNFy 组；主要功能是设置具体哪种输入输出的模式；例如如果是管脚输出，那么要确定是通用推挽输出模式，还是开漏输出模式，还是复用功能推挽输出模式，还是复用功能开漏输出模式；如果是管脚输入，是模拟输入模式还是浮空输入模式，还是上拉/下拉输入模式等。
6） 其他的寄存器也是这样查看表和状态，我们寄存器复位值是 0x4444 4444，十六制的 4 转换成二进制是 0100，即 CNF=01，MODE=00，我们可以查表知道 MODE=00 表示这个管脚复位后是输入模式，CNF=01 表示是浮空输入模式。

是不是看得有点头晕了，但是没有办法，我们就是通过改变这些寄存器的值来设置芯片管脚的，使得芯片管脚按照寄存器手册里的规定来进行相应的工作；可以是输出或输入，可以是高电平或是低电平（这个是另外一个寄存器来控制，大家可以对应看手册里的寄存器说明如下图 4-7），从而达到我们控制一个 LED 灯亮和灭；

7.2	GPIO寄存器描述	75
7.2.1	端口配置低寄存器(GPIOx_CRL) (x=A..E)	75
7.2.2	端口配置高寄存器(GPIOx_CRH) (x=A..E)	75
7.2.3	端口输入数据寄存器(GPIOx_IDR) (x=A..E)	76
7.2.4	端口输出数据寄存器(GPIOx_ODR) (x=A..E)	76
7.2.5	端口位设置/清除寄存器(GPIOx_BSRR) (x=A..E)	77
7.2.6	端口位清除寄存器(GPIOx_BRR) (x=A..E)	77
7.2.7	端口配置锁定寄存器(GPIOx_LCKR) (x=A..E)	77

图 4-7 GPIO 寄存器描述目录

像寄存器 GPIOx_CRH、GPIOx_IDR、GPIOx_ODR、GPIOx_BSRR、GPIOx_BRR、GPIOx_LCKR 都是同样的分析和阅读方法，我们接下来再来举例子详细说明。

4.4 分析一个最简单的程序

4.4.1 例程硬件原理图说明

这个例程 STM32 芯片的 PB8 管脚控制一个 LED 灯亮和灭的。

下面有个原理图（图 4-11）是用神舟 51+ARM 底板的 JP19 端连 LED 灯的负极，用正电源端连 LED 灯的正极，再串联一个限流电阻限制电流（电阻的作用就是限流、降压；如果线路上电阻很小，那么电压不变的情况下，电流就会变得很大，有可能会烧坏 LED 灯，所以这里我们串联一个电阻进行降压）降压防止 LED 灯被烧掉，这个串联电阻的阻值要计算好，使得在恒定电压的情况下，电流的大小刚好足够驱动 LED 灯点亮，点亮这个 LED 灯大概需要 10ma~20ma（毫安）的电流。

神舟 51+ARM 之 STM32F103C8T 的管脚如何控制这个 DS1 的灯呢？如图 4-8，我们可用一根杜邦线将 P0.0 和 JP19 上的 1 脚连起来(其实，连 JP19 上的任意一脚都可)。P0.0 实际上连的的是 STM32F103C8T6 的 PB8 管脚。可以看到，PB8 输出高电平的时候，LED 灯不会亮；只有当 PB8 输出低电平的时候，LED 灯才会点亮。所以我们想用 STM32F103C8T6 去驱动 DS1 这个 LED 灯亮，只要使得 PB8 输出低电平就可以，这样就知道如何控制这个 LED 灯了。

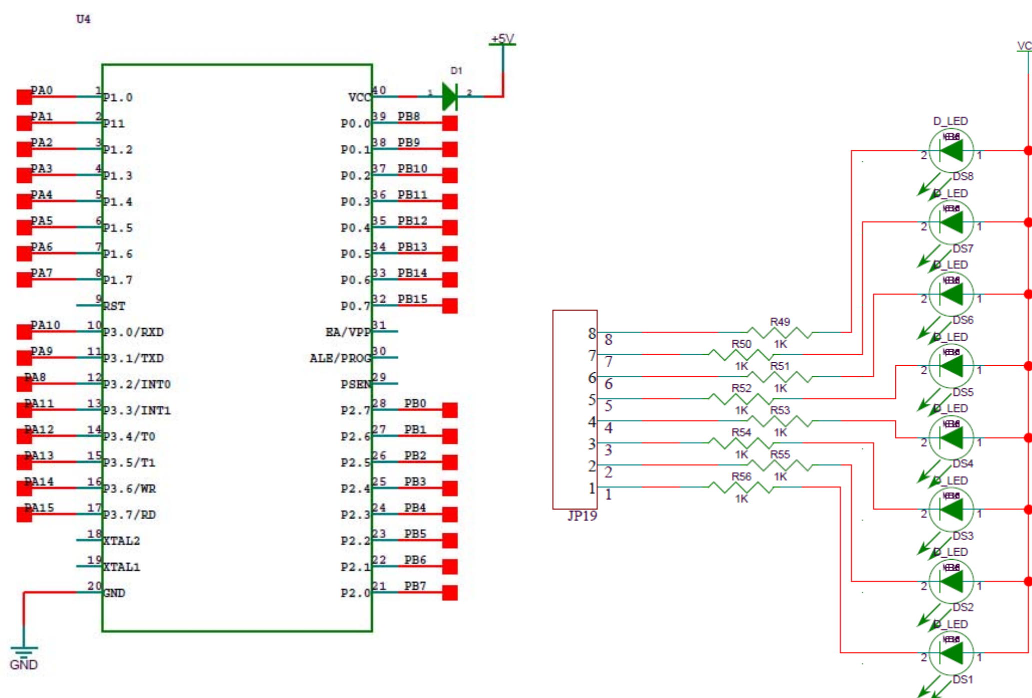


图 4-8 LED 电路原理图

为什么这么接呢？为什么不让 PB8 管脚接 LED 的正极，而 LED 灯的负极去接 GND 地呢？这样才是最常规的接法不对吗？答案是当然是，但是在这里这样的接法有助于芯片的长久使用，芯片的总体驱动能力是有限的，它可以驱动一个 LED 灯，但驱动不了 100 个，1000 个。

在这里需要重复上面已经说过的内容，首先我们要知道 LED 的发光工作条件，不同的 LED 其额定电压和额定电流不同，一般而言，红或绿颜色的 LED 的工作电压为 1.7V~2.4V，蓝或白颜色的 LED 工作电压为 2.7~4.2V， 直径为 3mm LED 的工作电流 2mA~10mA，在这里采用绿色的 LED；神舟 51+ARM 之 STM32F103C8T （如本实验板中所使用的 STM32F103C8T6 芯片）的 I/O 口作为输出口时，向外输出电流的能力是 25mA 左右，勉强是可以点亮一个发光二极管，但是如果我们用 STM32 去点亮很多个 LED 灯的时候，就有可能造成芯片本身输出电流不足(因为芯片能输出的总电流大小是恒定的)。

其次，PB8 的这种接法是一种灌电流（要 VCC 往内输入电流）的方式，这种方式使得 STM32 的芯片管脚让一个 LED 灯亮非常轻松，利用灌电流的方式驱动发光二极管是比较常见的一种用法，无论接多少 LED，芯片管脚的负荷都非常轻。当然，现今的一些增强型单片机，是采用拉电流输出的，只要单片机的输出电流能力足够强即可，不过接多了也是不可取的，单片机的总体驱动电流是有限的；上图中的电阻用的是 1K 阻值主要为了限制电流，让发光二极管的工作电流限定在 2mA~10mA。

4.4.2 例程main.c源代码（可以直接运行）：

以下是 main.c 的源文件，读者可以直接粘贴编译：

```
/*********** 此段代码直接拷贝进去可以直接运行 开始 *****************/
#define    __IO    volatile
```

```
typedef unsigned                    int uint32_t;
typedef __IO uint32_t    vu32;
typedef unsigned short         int uint16_t;

#define GPIO_Pin_0                        ((uint16_t)0x0001)    /*!< Pin 0 selected */
#define GPIO_Pin_1                        ((uint16_t)0x0002)    /*!< Pin 1 selected */
#define GPIO_Pin_2                        ((uint16_t)0x0004)    /*!< Pin 2 selected */
#define GPIO_Pin_3                        ((uint16_t)0x0008)    /*!< Pin 3 selected */
#define GPIO_Pin_4                        ((uint16_t)0x0010)    /*!< Pin 4 selected */
#define GPIO_Pin_5                        ((uint16_t)0x0020)    /*!< Pin 5 selected */
#define GPIO_Pin_6                        ((uint16_t)0x0040)    /*!< Pin 6 selected */
#define GPIO_Pin_7                        ((uint16_t)0x0080)    /*!< Pin 7 selected */
#define GPIO_Pin_8                        ((uint16_t)0x0100)    /*!< Pin 8 selected */
#define GPIO_Pin_9                        ((uint16_t)0x0200)    /*!< Pin 9 selected */
#define GPIO_Pin_10                       ((uint16_t)0x0400)    /*!< Pin 10 selected */
#define GPIO_Pin_11                       ((uint16_t)0x0800)    /*!< Pin 11 selected */
#define GPIO_Pin_12                       ((uint16_t)0x1000)    /*!< Pin 12 selected */
#define GPIO_Pin_13                       ((uint16_t)0x2000)    /*!< Pin 13 selected */
#define GPIO_Pin_14                       ((uint16_t)0x4000)    /*!< Pin 14 selected */
#define GPIO_Pin_15                       ((uint16_t)0x8000)    /*!< Pin 15 selected */
#define GPIO_Pin_All                      ((uint16_t)0xFFFF)    /*!< All pins selected */

#define RCC_APB2Periph_AFIO              ((uint32_t)0x00000001)
#define RCC_APB2Periph_GPIOA             ((uint32_t)0x00000004)
#define RCC_APB2Periph_GPIOB             ((uint32_t)0x00000008)

/************ GPIOB <*************/
typedef struct
{
  __IO uint32_t CRL;
  __IO uint32_t CRH;
  __IO uint32_t IDR;
  __IO uint32_t ODR;
  __IO uint32_t BSRR;
  __IO uint32_t BRR;
  __IO uint32_t LCKR;
} GPIO_TypeDef;

typedef struct
{
  __IO uint32_t CR;
  __IO uint32_t CFGR;
  __IO uint32_t CIR;
  __IO uint32_t APB2RSTR;
```

```
  __IO uint32_t APB1RSTR;
  __IO uint32_t AHBENR;
  __IO uint32_t APB2ENR;
  __IO uint32_t APB1ENR;
  __IO uint32_t BDCR;
  __IO uint32_t CSR;
} RCC_TypeDef;

/********* GPIOB 管脚的内存对应地址  *******/
#define PERIPH_BASE                    ((uint32_t)0x40000000)
#define APB2PERIPH_BASE              (PERIPH_BASE + 0x10000)
#define GPIOB_BASE                     (APB2PERIPH_BASE + 0x0C00)
#define GPIOB                        ((GPIO_TypeDef *) GPIOB_BASE)

/************ RCC 时钟  <*************/
#define AHBPERIPH_BASE               (PERIPH_BASE + 0x20000)
#define RCC_BASE                       (AHBPERIPH_BASE + 0x1000)
#define RCC                          ((RCC_TypeDef *) RCC_BASE)


/************ www.armjishu.com *******/
void Delay(vu32 nCount);

int main(void)   //main 是程序入口
{
     /*  使能 APB2 总线的时钟，对 GPIO 的端口 B 时钟使能  */
     RCC->APB2ENR |= RCC_APB2Periph_GPIOB;

     /*-- GPIO Mode Configuration 速度，输入或输出  -----------------------*/
     /*-- GPIO CRL Configuration  设置 IO 端口低 8 位的模式（输入还是输出）---*/
     /*-- GPIO CRH Configuration  设置 IO 端口高 8 位的模式（输入还是输出）---*/
     GPIOB->CRH &=    0xFFFFFFF0;
     GPIOB->CRH |=    0x00000003;

     while (1)
     {
          GPIOB->BRR = GPIO_Pin_8;
          Delay(0x2FFFFF);
          GPIOB->BSRR = GPIO_Pin_8;
          Delay(0x2FFFFF);
     }
}

void Delay(vu32 nCount)              //通过不断 for 循环 nCount 次，达到延时的目的口
```

```
{
   for(; nCount != 0; nCount--);
}
/*********** 此段代码直接拷贝进去可以直接运行 结束 *************/
```

4.4.3 例程环境搭建

1. 点击桌面UVision4图标，启动软件，如下图4-9。如果是第一次使用的话会打开一个自带的工程文件，我们可以通过工具栏Project->Close Project选项把它关掉。

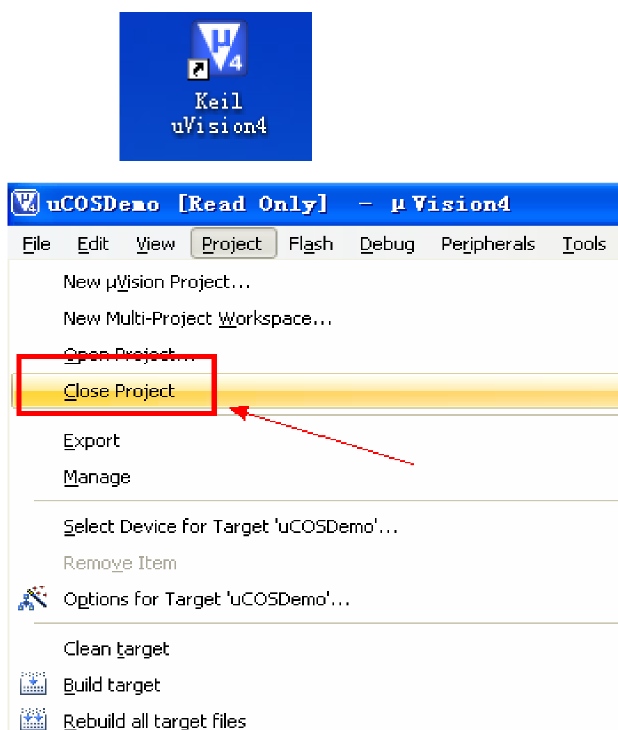


图 4-9 启动 KEIL 并关掉当前的工程

2. 在工具栏Project->New µVision Project...新建我们的工程文件，如图4-10，我们将新建的工程文件保存在桌面的"神舟51+ARM之STM32F103C8T开发板模板工程"（先在电脑桌面上新建一个"神舟51+ARM之STM32F103C8T开发板模板工程"文件夹，在该文件夹里面新建一个Project文件夹），文件名取为：STM32-DEMO（英文DEMO的意思是例子），名字可以随取，点击保存，如图4-11。

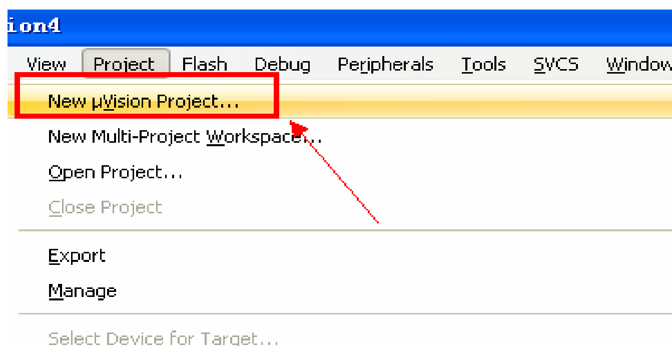


图 4-10 新建工程文件

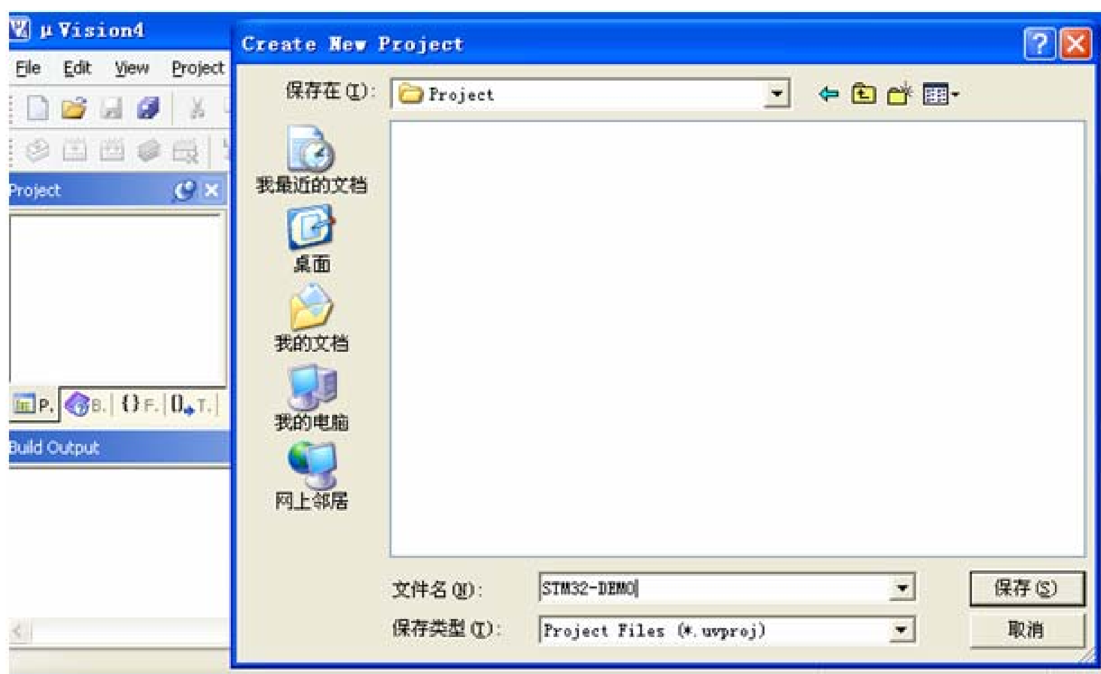


图 4.11 选择新建工程文件保存的位置

3. 接下来的窗口是让我们选择公司跟芯片的型号，因为我们用的芯片是 ST 公司的 STM32F103C8T6，有 20K SRAM,64K Flash，属于高集成度的芯片。按如下选择即可如图 4-12、4-13。

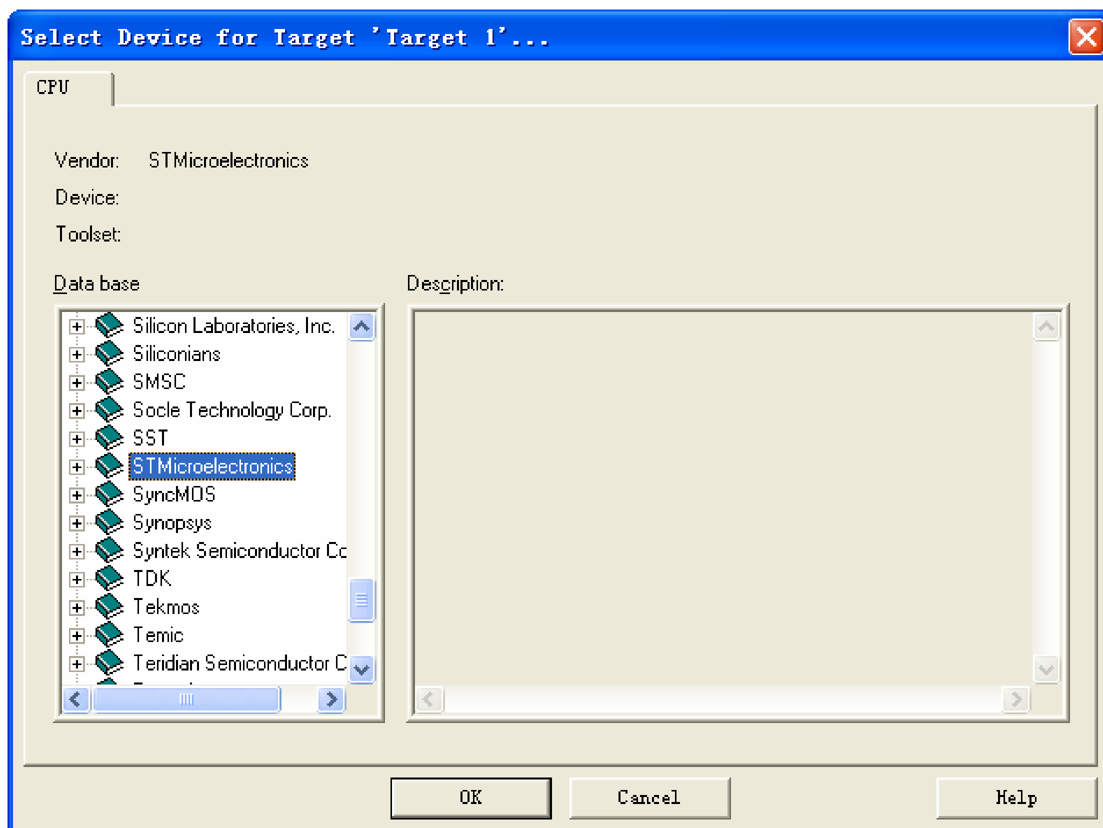


图 4-12 选择 STM32 处理器芯片

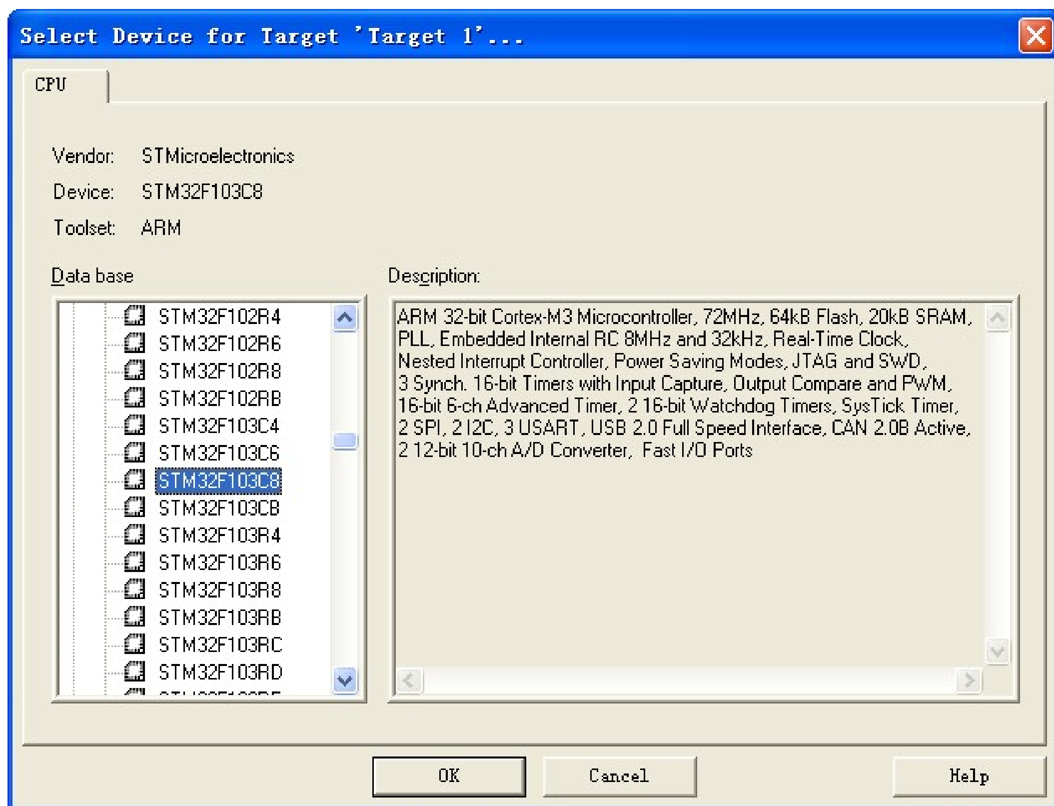


图 4-13 选择 STM32 处理器芯片

4. 接下来的窗口问我们是否需要拷贝STM32的启动代码到工程文件中，如图4-14，这份启动代码在M3系列中都是适用的，一般情况下我们都点击是，但我们这里用的是ST的库，库文件里面也自带了这一份启动代码，所以为了保持库的完整性，我们就不需要开发环境为我们自带的启动代码了，稍后我们自己手动添加，这里我们点击否。

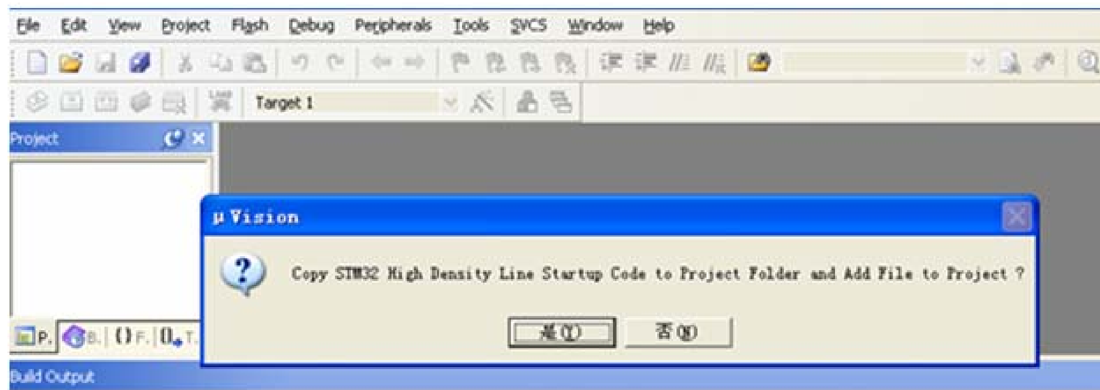


图 4-14 启动代码的选择

5. 此时我们的工程新建成功，如下图4-15所示。但我们的工程中还没有任何文件，接下来我们需要在我们的工程中添加所需文件。

STM32-DEMO.plg
STM32-DEMO.uvproj
STM32F10x.s

图 4-15 新建工程完成

6．可以看到目前工程里只有一个文件如下图 4-16 所示：

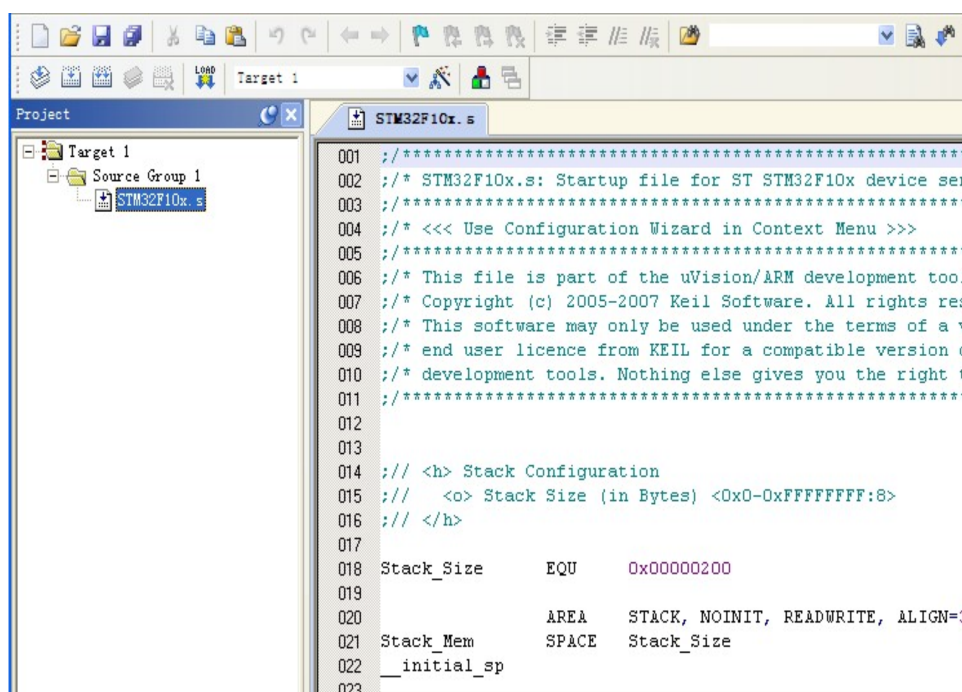


图 4-16 新建工程完成页面

7．新建一个 main.c 文件存放在路径：神舟 51+ARM 之 STM32F103C8T 开发板模板工程\Project\下，然后按照以下图标操作过程把 main.c 文件添加到工程里如图 4-17、4-18 所示：

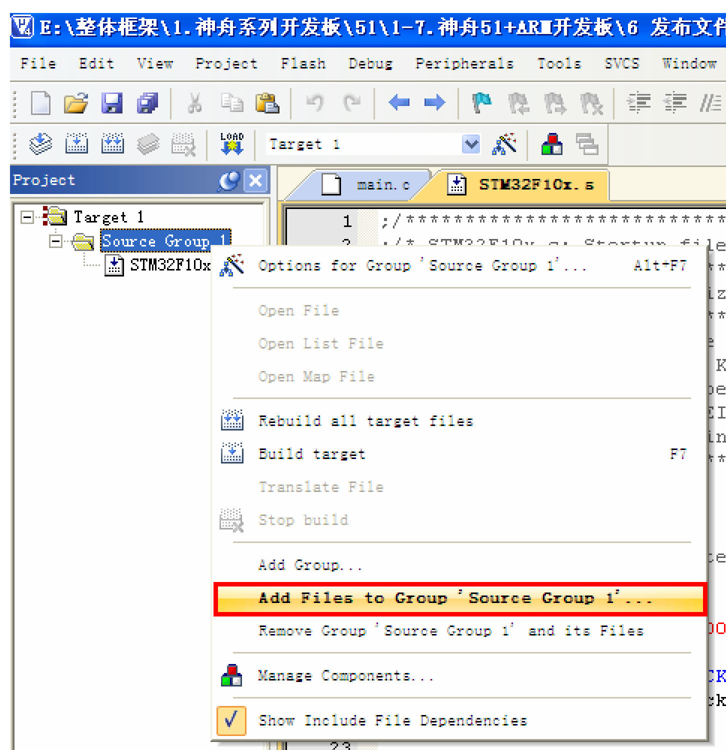


图 4-17 添加新建的 main.c 文件到工程中

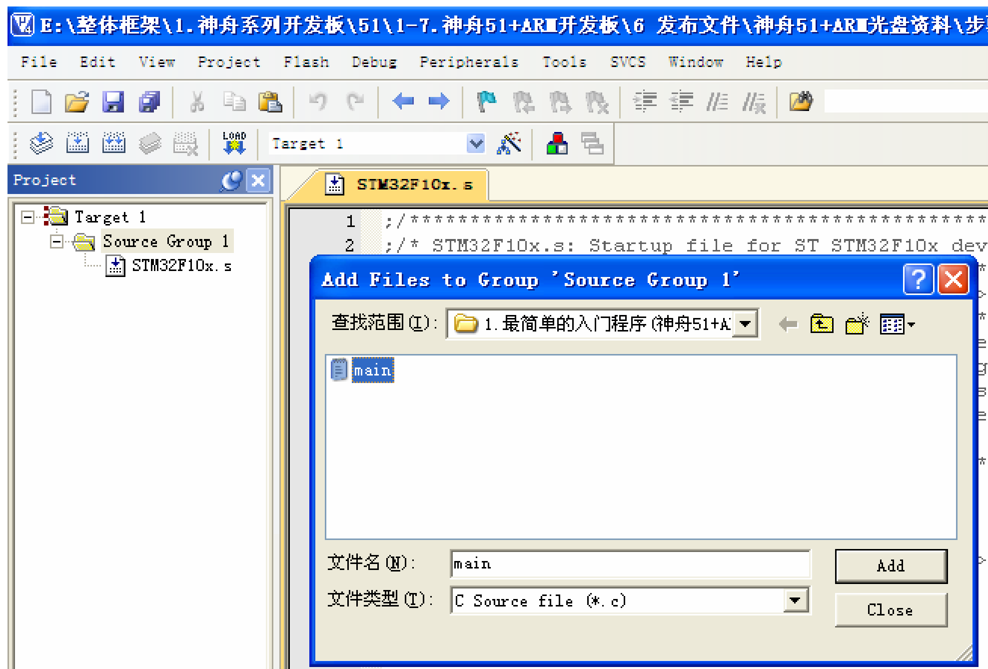


图 4-18 加新建的 main.c 文件到工程中

这个例程，我们将所有的代码都写到了一个 mian.c 文件，不涉及到任何库函数，也没有包含任何的头文件，下图 4-19 为我们的截图：

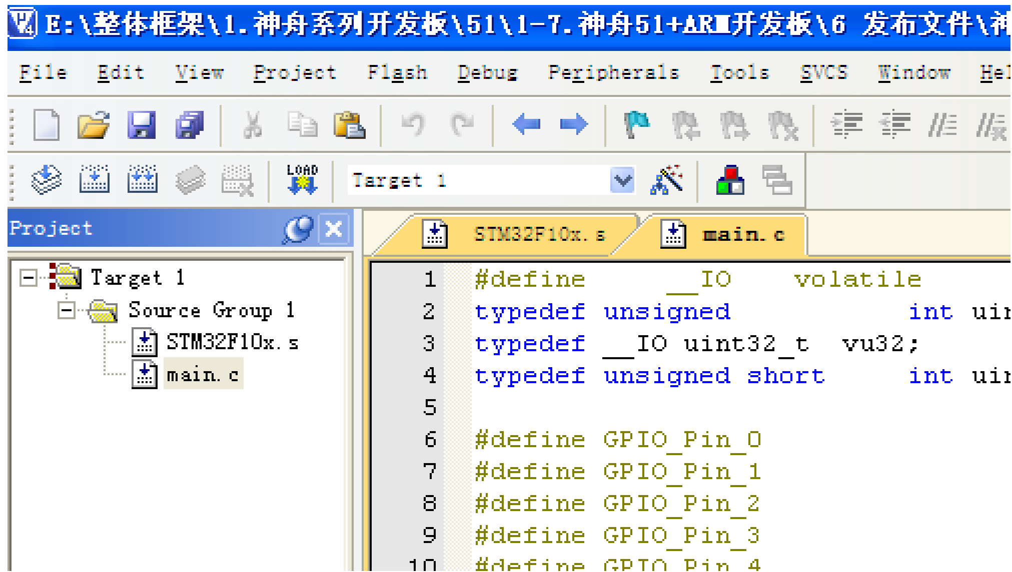


图 4-19 添加完 main 文件

8．把 5.61 节的代码直接全盘拷贝到 main.c 文件里，如下图 4-20

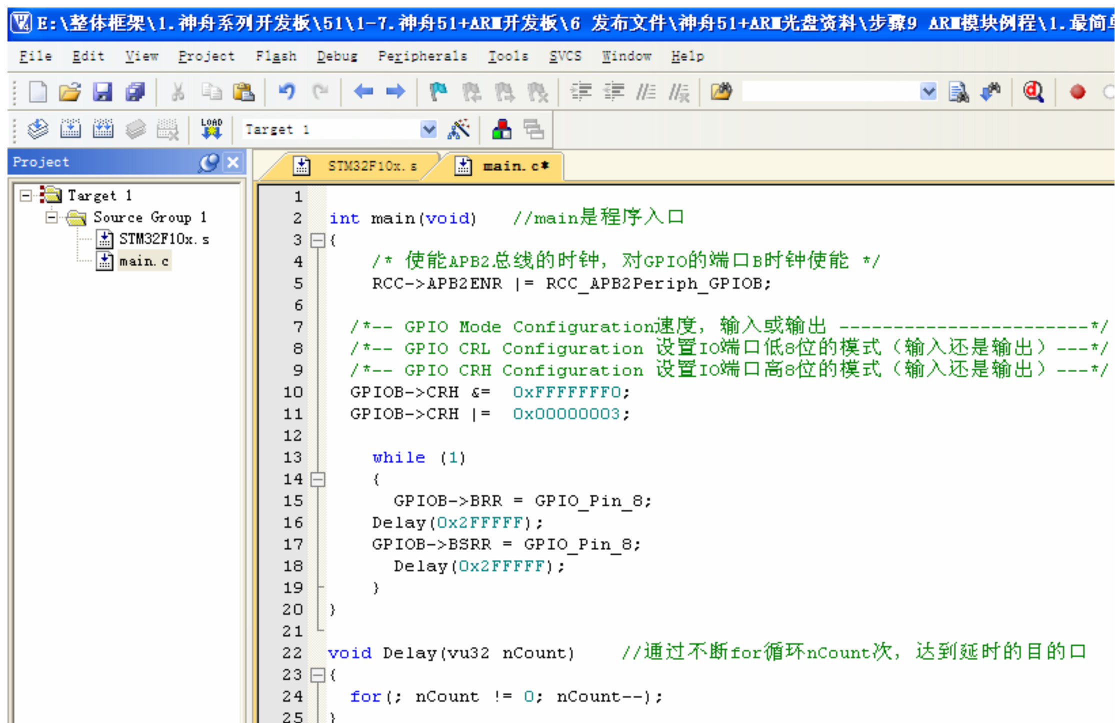


图 4-20 在 main.c 文件上有我们的程序

9．点一下按钮，可以看到编译成功，如下图 4-21 所示。我们可以看到编译后的 HEX 文件，我们可以直接在光盘中找到这个文件，直接进入 Project 文件夹，打开即可，如下图 4-22 所示：

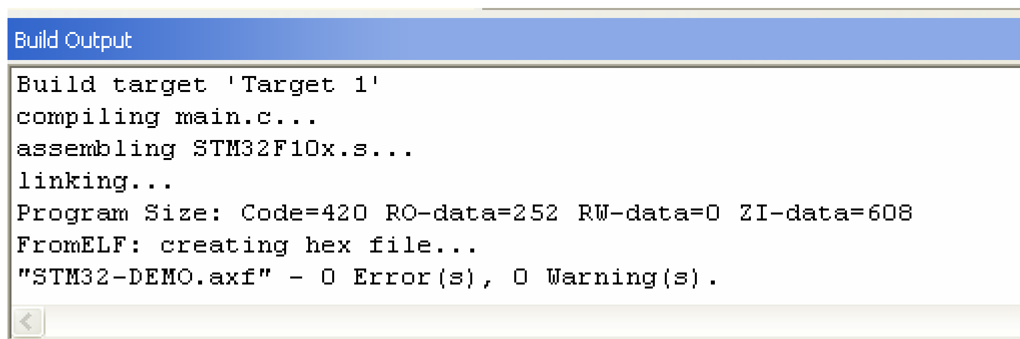


图 4-21 程序编译完成



图 4-22 查找编译后的文件

10. 该代码可以直接下载到神舟 51+ARM 开发板中，按一下复位按键，可以看到 LED 灯一亮一灭，具体下载方式我们推荐有三种，具体下载设置请参考手册其他章节：

1）JLINK V8 仿真器下载（我们推荐）

2）ULINK2 仿真器下载

3）串口下载

4.4.4 实验现象

可以看神舟 51+ARM 开发板的 DS1 灯一亮一灭的闪烁。

4.4.5 例程软件架构和代码分析（只有一个main.c文件）

1. 第一部分的代码分析：

```
#define        __IO        volatile
typedef unsigned                    int uint32_t;
typedef __IO uint32_t    vu32;
typedef unsigned short          int uint16_t;
```

分析 1：volatile 是什么？怎么用？

答：简单的说，就是不让编译器进行优化，即每次读取或者修改值的时候，都必须重新从内存或者寄存器中读取或者修改，防止从缓存处读取的值是过期了的，所以加了这个 volatile 可以保证每次读的值绝对是实时的：

一般说来，volatile 用在如下的几个地方：

1. 中断服务程序中修改的供其它程序检测的变量需要加 volatile；
2. 多任务环境下各任务间共享的标志应该加 volatile；
3. 存储器映射的硬件寄存器通常也要加 volatile 说明，因为每次对它的读写都可能由不同意义。

我认为这是区分 C 程序员和嵌入式系统程序员的最基本的问题。搞嵌入式的家伙们经常同硬件、中断、RTOS 等等打交道，所有这些都要求用到 volatile 变量。不懂得 volatile

的内容将会带来灾难。假设被面试者正确地回答了这是问题（嗯，怀疑是否会是这样），我将稍微深究一下，看一下这家伙是不是直正懂得 volatile 完全的重要性。

分析 2：__I、 __O 、__IO 是什么？

答：如下：

__I ：输入口。既然是输入，那么寄存器的值就随时会外部修改，那就不能进行优化，每次都要重新从寄存器中读取。也不能写，即只读，不然就不是输入而是输出了。

__O ：输出口，也不能进行优化，不然你连续两次输出相同值，编译器认为没改变，就忽略了后面那一次输出，假如外部在两次输出中间修改了值，那就影响输出。

__IO：输入输出口，同上。

分析 3：为什么加下划线？

答：原因是避免命名冲突，一般宏定义都是大写，但因为这里的字母比较少，所以再添加下划线来区分。这样一般都可以避免命名冲突问题，因为很少人这样命名，这样命名的人肯定知道这些是有什么用的。

经常写大工程时，都会发现老是命名冲突，要不是全局变量冲突，要不就是宏定义冲突，所以我们要尽量避免这些问题，不然出问题了都不知道问题在哪里。

分析 4：typedef 是什么意思，怎么使用？

答：typedef 为 C 语言的关键字，作用是为一种数据类型定义一个新名字。这里的数据类型包括内部数据类型（int,char 等）和自定义的数据类型（struct 等）；在编程中使用 typedef 目的一般有两个，一个是给变量一个易记且意义明确的新名字，另一个是简化一些比较复杂的类型声明。

1） typedef 的最简单使用，例如:typedef long byte_4;表示给已知数据类型 long 起个新名字，叫 byte_4

2） typedef 与结构结合使用

例如：typedef struct tagMyStruct

```
{
int iNum;
long lLength;
} MyStruct;
```

这语句实际上完成两个操作

操作 1：定义一个新的结构类型 tagMyStruct, struct 关键字和 tagMyStruct 一起，构成了这个结构类型，不论是否有 typedef，这个结构都存在。我们可以用 struct tagMyStruct varName 来定义变量，但要注意，使用 tagMyStruct varName 来定义变量是不对的，因为 struct 和 tagMyStruct 合在一起才能表示一个结构类型。

```
struct tagMyStruct
{
int iNum;
long lLength;
};
```

操作 2：typedef 为这个新的结构起了一个名字，叫 MyStruct。

```
typedef struct tagMyStruct MyStruct;
```

因此，MyStruct 实际上相当于 struct tagMyStruct，我们可以使用 MyStruct

varName 来定义变量。

分析 5：所以具体的 typedef 代码解释如下：
1）例：typedef unsigned int uint32_t;
表示使用 uint32_t 符号表示 unsigned int 符号
2）例：typedef __IO uint32_t vu32;
表示使用 vu32 符号表示 typedef __IO 符号
3）例：typedef unsigned short int uint16_t;
表示使用 uint16_t 符号表示 unsigned short int 符号

2.初始化宏定义详细分析分解：

```
#define GPIO_Pin_0                 ((uint16_t)0x0001)  /*!< Pin 0 selected */
#define GPIO_Pin_1                 ((uint16_t)0x0002)  /*!< Pin 1 selected */
#define GPIO_Pin_2                 ((uint16_t)0x0004)  /*!< Pin 2 selected */
#define GPIO_Pin_3                 ((uint16_t)0x0008)  /*!< Pin 3 selected */
#define GPIO_Pin_4                 ((uint16_t)0x0010)  /*!< Pin 4 selected */
#define GPIO_Pin_5                 ((uint16_t)0x0020)  /*!< Pin 5 selected */
#define GPIO_Pin_6                 ((uint16_t)0x0040)  /*!< Pin 6 selected */
#define GPIO_Pin_7                 ((uint16_t)0x0080)  /*!< Pin 7 selected */
#define GPIO_Pin_8                 ((uint16_t)0x0100)  /*!< Pin 8 selected */
#define GPIO_Pin_9                 ((uint16_t)0x0200)  /*!< Pin 9 selected */
#define GPIO_Pin_10                ((uint16_t)0x0400)  /*!< Pin 10 selected */
#define GPIO_Pin_11                ((uint16_t)0x0800)  /*!< Pin 11 selected */
#define GPIO_Pin_12                ((uint16_t)0x1000)  /*!< Pin 12 selected */
#define GPIO_Pin_13                ((uint16_t)0x2000)  /*!< Pin 13 selected */
#define GPIO_Pin_14                ((uint16_t)0x4000)  /*!< Pin 14 selected */
#define GPIO_Pin_15                ((uint16_t)0x8000)  /*!< Pin 15 selected */
#define GPIO_Pin_All               ((uint16_t)0xFFFF)  /*!< All pins selected */
#define RCC_APB2Periph_AFIO              ((uint32_t)0x00000001)
#define RCC_APB2Periph_GPIOA             ((uint32_t)0x00000004)
#define RCC_APB2Periph_GPIOB             ((uint32_t)0x00000008)
/************ GPIOB <*************/
typedef struct
{
  __IO uint32_t CRL;
  __IO uint32_t CRH;
  __IO uint32_t IDR;
  __IO uint32_t ODR;
  __IO uint32_t BSRR;
  __IO uint32_t BRR;
  __IO uint32_t LCKR;
} GPIO_TypeDef;
typedef struct
{
```

```
    __IO uint32_t CR;
    __IO uint32_t CFGR;
    __IO uint32_t CIR;
    __IO uint32_t APB2RSTR;
    __IO uint32_t APB1RSTR;
    __IO uint32_t AHBENR;
    __IO uint32_t APB2ENR;
    __IO uint32_t APB1ENR;
    __IO uint32_t BDCR;
    __IO uint32_t CSR;
} RCC_TypeDef;
/********* GPIOB 管脚的内存对应地址 *******/
#define PERIPH_BASE          ((uint32_t)0x40000000)
#define APB2PERIPH_BASE      (PERIPH_BASE + 0x10000)
#define GPIOB_BASE           (APB2PERIPH_BASE + 0x0C00)
#define GPIOB                ((GPIO_TypeDef *) GPIOB_BASE)
/************ RCC 时钟 <*************/
#define AHBPERIPH_BASE       (PERIPH_BASE + 0x20000)
#define RCC_BASE             (AHBPERIPH_BASE + 0x1000)
#define RCC                  ((RCC_TypeDef *) RCC_BASE)
```

分析 1：可以看出规律，GPIO_Pin_0、GPIO_Pin_1 到 GPIO_Pin_15 总共 16 个 define 定义每个都是一个 16 比特(uint16_t)的对象。

每个端口都有 16 个 GPIO 管脚，比如 GPIOA，GPIOB，GPIOC 等，我们用 16bit 的位来表示，即 2 个字节，每个 bit 表示 16 个 GPIO 管脚中的一个，可以看下面 2 个寄存器，就是控制 GPIO 对应的具体管脚是高电平还是低电平的，通过对设置对应位为 0 或为 1，就可以使得管脚的电平为高或低。

每个 GPIO_Pin_x 占用这 16 个比特中的 1 个位，其他剩余的 15 个位都是 0，这 16 个 GPIO_Pin_x 就被用来表示芯片各个不同端口的 16 个管脚，比如 PA0、PA1 一直到 PA15 分别对应 GPIO_Pin_0、GPIO_Pin_1 到 GPIO_Pin_15，这样定义好之后具体如何使用我们后面还会再说。

```
#define GPIO_Pin_0     ((uint16_t)0x0001)    0000 0000 0000 0001
#define GPIO_Pin_1     ((uint16_t)0x0002)    0000 0000 0000 0010
#define GPIO_Pin_2     ((uint16_t)0x0004)    0000 0000 0000 0100
#define GPIO_Pin_3     ((uint16_t)0x0008)    0000 0000 0000 1000
#define GPIO_Pin_4     ((uint16_t)0x0010)    0000 0000 0001 0000
#define GPIO_Pin_5     ((uint16_t)0x0020)    0000 0000 0010 0000
#define GPIO_Pin_6     ((uint16_t)0x0040)    0000 0000 0100 0000
#define GPIO_Pin_7     ((uint16_t)0x0080)    0000 0000 1000 0000
#define GPIO_Pin_8     ((uint16_t)0x0100)    0000 0001 0000 0000
#define GPIO_Pin_9     ((uint16_t)0x0200)    0000 0010 0000 0000
#define GPIO_Pin_10    ((uint16_t)0x0400)    0000 0100 0000 0000
#define GPIO_Pin_11    ((uint16_t)0x0800)    0000 1000 0000 0000
#define GPIO_Pin_12    ((uint16_t)0x1000)    0001 0000 0000 0000
```

```
#define GPIO_Pin_13        ((uint16_t)0x2000)     0010 0000 0000 0000
#define GPIO_Pin_14        ((uint16_t)0x4000)     0100 0000 0000 0000
#define GPIO_Pin_15        ((uint16_t)0x8000)     1000 0000 0000 0000
#define GPIO_Pin_All       ((uint16_t)0xFFFF)     1111 1111 1111 1111
```

分析 2：这里是定义 GPIO 端口 B 的一些初始化变量，后面那些具体的地址需要查看 STM32 中文参考手册中的 2.3 节 存储器映像表，如图 4-25 所示

```
/********* GPIOB 管脚的内存对应地址 *******/
#define PERIPH_BASE              ((uint32_t)0x40000000)
#define APB2PERIPH_BASE          (PERIPH_BASE + 0x10000)
#define GPIOB_BASE               (APB2PERIPH_BASE + 0x0C00)
#define GPIOB                    ((GPIO_TypeDef *) GPIOB_BASE)
```

0X4001 0C00 - 0x4001 0FFF	GPIO端口B
0x4001 0800 - 0x4001 0BFF	GPIO端口A
0x4001 0400 - 0x4001 07FF	EXTI
0x4001 0000 - 0x4001 03FF	AFIO

图 4-23 GPIO 端口地址

1）定义总线的基地址（这个需要参考手册）

```
#define    PERIPH_BASE      ((uint32_t)0x40000000)
```

2）APB2PERIPH_BASE（APB2 时钟总线）的地址是在总线基地址上加多 0x10000，刚好就是上图的 AFIO 寄存器地址，具体可以看参考手册

```
#define    APB2PERIPH_BASE      (PERIPH_BASE + 0x10000)
```

3）定义 GPIO 端口 B 的基地址，该地址是 0x40001 0c00。

```
#define    GPIOB_BASE      (APB2PERIPH_BASE + 0x0C00)
```

4）定义一个 GPIO_TypeDef 的 struct 结构，从 GPIO 端口 B 的基地址开始进行覆盖

```
#define    GPIOB      ((GPIO_TypeDef *)    GPIOB_BASE)
```

分析 3：这里是初始化 RCC 时钟总线的基地址，如图 4-24 所示，详细分析与上面同原理，具体的地址需要查看芯片参考手册

```
/************ RCC 时钟 <*************/
#define AHBPERIPH_BASE         (PERIPH_BASE + 0x20000)
#define RCC_BASE               (AHBPERIPH_BASE + 0x1000)
#define RCC                    ((RCC_TypeDef *) RCC_BASE)
```

0x4002 3000 - 0x4002 33FF	CRC
0x4002 2000 - 0x4002 23FF	闪存存储器接口
0x4002 1400 - 0x4002 1FFF	保留
0x4002 1000 - 0x4002 13FF	复位和时钟控制(RCC)
0x4002 0800 - 0x4002 0FFF	保留
0x4002 0400 - 0x4002 07FF	DMA2
0x4002 0000 - 0x4002 03FF	DMA1

图 4-24 RCC 时钟总线基地址

分析 4：设置端口的偏移量，后面我们会详细解释

```
#define RCC_APB2Periph_AFIO          ((uint32_t)0x00000001)
#define RCC_APB2Periph_GPIOA         ((uint32_t)0x00000004)
#define RCC_APB2Periph_GPIOB         ((uint32_t)0x00000008)
```

3. 定义这两个结构体与芯片参考手册中的寄存器进行对应，芯片参考手册中对应的寄存器都是 32bit 的，所以在这个结构体的各个对象都被定义成 uint32_t 类型，并且是__IO 类型，表示每次操作寄存器都是实时获取数据，如下图 4-25：

```
40 typedef struct
41 {
42     __IO uint32_t CRL;
43     __IO uint32_t CRH;
44     __IO uint32_t IDR;
45     __IO uint32_t ODR;
46     __IO uint32_t BSRR;
47     __IO uint32_t BRR;
48     __IO uint32_t LCKR;
49 } GPIO_TypeDef;
50
```

```
51 typedef struct
52 {
53     __IO uint32_t CR;
54     __IO uint32_t CFGR;
55     __IO uint32_t CIR;
56     __IO uint32_t APB2RSTR;
57     __IO uint32_t APB1RSTR;
58     __IO uint32_t AHBENR;
59     __IO uint32_t APB2ENR;
60     __IO uint32_t APB1ENR;
61     __IO uint32_t BDCR;
62     __IO uint32_t CSR;
63 } RCC_TypeDef;
```

图 4-25 结构图

4. 下图 4-26 是 main 函数的剖析，总共来说分为 4 个步骤，下面一一介绍：

```
int main(void)    //main是程序入口
{
    /* 使能APB2总线的时钟，对GPIO的端口B时钟使能 */
    RCC->APB2ENR |= RCC_APB2Periph_GPIOB;

    /*-- GPIO Mode Configuration速度，输入或输出 -----------------------*/
    /*-- GPIO CRL Configuration 设置IO端口低8位的模式（输入还是输出）---*/
    /*-- GPIO CRH Configuration 设置IO端口高8位的模式（输入还是输出）---*/
    GPIOB->CRH &=  0xFFFFFFF0;
    GPIOB->CRH |=  0x00000003;

    while (1)
    {
        GPIOB->BRR = GPIO_Pin_8;
        Delay(0x2FFFFF);
        GPIOB->BSRR = GPIO_Pin_8;
        Delay(0x2FFFFF);
    }
}
```

第一步

第二步

第三步

第四步

图 4-26 main 函数的剖析

步骤 1：使能 APB2 总线的时钟。

对 GPIO 的端口 B 时钟使能，这个是芯片厂家所规定的操作，我们先按照这样来操作就可以，具体实现方式也是将对应 GPIOB 寄存器使能，同时，也有 RCC，串口接口，CAN 接口，485 接口等时钟的使能寄存器，使用前都需要先对时钟总线使能的。

使能操作完毕，就相当于我们对要使用的这个接口功能进行使能和激活，每个接口在使用前都必须要求使能和激活，只有激活后才可以使用。

步骤 2：配置 GPIO 端口的状态。

输入还是输出，速度多少，和大家可以参考对应的芯片寄存器手册，可以看到我们将 PB2 设置成‘00：通用推挽输出模式’并且速度是‘11：输出模式，最大速度 50MHz’

具体可以参考 GPIOB_CRL 寄存器，我们将这个寄存器设置为了 0x0000 0300。

步骤 3：进入 while 死循环

可以使得我们的点灯程序一直不会退出，达到重复一亮一灭的功能。

步骤 4：GPIO 输入和输出使得灯亮灭

GPIOB_BSRR 对应位设置 1，可以使得对应管脚的 ODR 位为 1；GPIO_BRR 的对应位设置 1，可以使得对应管脚的 ODR 位为 0

那么 ODR 位是什么呢？这个就是端口输出数据寄存器 GPIOB_ODR，实际上这个寄存器里的对应位的变化才是真正 GPIO 管脚高低电平的变化；而寄存器 GPIOB_BSRR 和寄存器 GPIOB_BRR 则可以间接影响到它。

当然上面程序，我们也可以改成直接操作 GPIOB_ODR 的代码，如下表 4-2，这个留个大家来做练习吧。

当然上面程序，我们也可以改成直接操作 GPIOB_ODR 的代码，GPIOx_ODR 寄存器结构与说明如下表 4-2、4-3 所示，GPIOx_ODR 的地址偏移值为 0x0C，复位值为 0x00.这个留个大家来做练习吧。

表 4-2　GPIOx_ODR 寄存器

BIT 31	BIT 30	BIT 29	BIT 28	BIT 27	BIT 26	BIT 25	BIT 24
保留							

BIT 23	BIT 22	BIT 21	BIT 20	BIT 19	BIT 18	BIT 17	BIT 16
保留							

BIT 15	BIT 14	BIT 13	BIT 12	BIT 11	BIT 10	BIT 9	BIT 8
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8
rw	rw	rw	rw	rw	rw	rw	rw

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT0
ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw

表 4-3 GPIOB_ODR 寄存器低 16 位的功能配置

位 31:16	保留
位15:0	**ODRy[15:0]**：端口输出数据(y = 0…15) 这些位可读可写并只能以字(16位)的形式操作。 注：对 GPIOx_BSRR(x = A…E)，可以分别地对各个 ODR 位进行独立的设置/清除。

4.4.6 代码剖析 1---代码的定义如何与芯片内部资源挂钩

C 语言程序代码如何真正访问芯片内部寄存器的呢？大家看下面这些定义：

```
/********* GPIOB 管脚的内存对应地址  *******/
#define PERIPH_BASE              ((uint32_t)0x40000000)
#define APB2PERIPH_BASE           (PERIPH_BASE + 0x10000)
#define GPIOB_BASE               (APB2PERIPH_BASE + 0x0C00)
#define GPIOB                  ((GPIO_TypeDef *) GPIOB_BASE)

/************ RCC 时钟  <*************/
#define AHBPERIPH_BASE            (PERIPH_BASE + 0x20000)
#define RCC_BASE                 (AHBPERIPH_BASE + 0x1000)
#define RCC                    ((RCC_TypeDef *) RCC_BASE)
```

通过这几个 define 可以算出来一下地址如下图 4-27：

GPIOB = 0x4000 0000 + 0x1 0000 + 0x0C00 = 0x4001 0C00 刚好与 PortB 在内存中的位置对应上

RCC = 0x4000 0000 + 0x2 0000 + 0x1000 = 0x4002 1000 刚好也与 RCC 在内存中的位置对应上

```
0x4001 1400                 0x4002 2000
              Port C                      reserved
0x4001 1000                 0x4002 1400
              Port B                      RCC
0x4001 0C00                 0x4002 1000
              Port A                      reserved
0x4001 0800                 0x4002 0400
```

图 4-27 地址的计算

更多内存映射可以找到《STM32F103xB 的数据手册》，如下图 4-28，可以详细的进行了解了解：

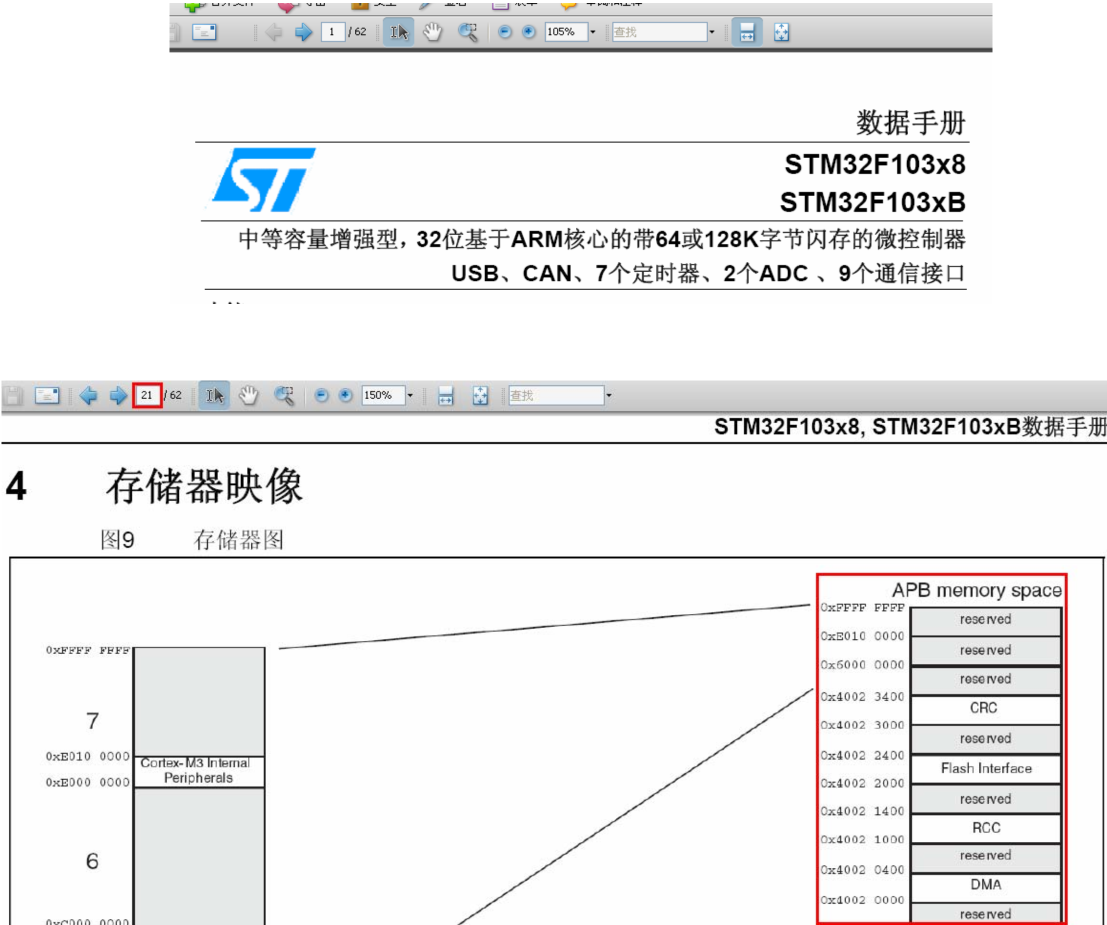


图 4-28 STM32F103XB 数据手册资料

通过以上这个存储器映像图，我们就可以通过代码与存储器地址关联起来了。

4.4.7 代码剖析 2---代码如何映射到芯片内部的寄存器

这也是从库函数中摘抄出来的一种实现方式，我们通过 struct 结构可以完成对 GPIO，RCC 等外设模块中各个寄存器的管理，比如，一个 GPIO 模块中，有很多个寄存器，我们可以用 C 语言中的 struct 来对应这些寄存器。

在芯片中，一个寄存器连着一个寄存器，每个寄存器都是 32 位的（4 个字节）；我们在 struct 结构中的成员每个也都是 32 位的，一个连着一个，刚好一一对应，大家可以看到代码中对应的定义与《STM32 中文参考手册》的章节可以一一对应上，可以看下 4-29 图和 4-30 图：

```
typedef struct
{
  __IO uint32_t CRL;
  __IO uint32_t CRH;
  __IO uint32_t IDR;
  __IO uint32_t ODR;
  __IO uint32_t BSRR;
  __IO uint32_t BRR;
  __IO uint32_t LCKR;
} GPIO_TypeDef;
```

7.2 GPIO寄存器描述

7.2.1 端口配置低寄存器(GPIOx_CRL) (x=A..E)
7.2.2 端口配置高寄存器(GPIOx_CRH) (x=A..E)
7.2.3 端口输入数据寄存器(GPIOx_IDR) (x=A..E)
7.2.4 端口输出数据寄存器(GPIOx_ODR) (x=A..E)
7.2.5 端口位设置/清除寄存器(GPIOx_BSRR) (x=A..E)
7.2.6 端口位清除寄存器(GPIOx_BRR) (x=A..E)
7.2.7 端口配置锁定寄存器(GPIOx_LCKR) (x=A..E)

图 4-29 GPIO 文档解释与代码对应章节图

```
typedef struct
{
  __IO uint32_t CR;
  __IO uint32_t CFGR;
  __IO uint32_t CIR;
  __IO uint32_t APB2RSTR;
  __IO uint32_t APB1RSTR;
  __IO uint32_t AHBENR;
  __IO uint32_t APB2ENR;
  __IO uint32_t APB1ENR;
  __IO uint32_t BDCR;
  __IO uint32_t CSR;
} RCC_TypeDef;
```

6.3 RCC寄存器描述

6.3.1 时钟控制寄存器(RCC_CR)
6.3.2 时钟配置寄存器(RCC_CFGR)
6.3.3 时钟中断寄存器 (RCC_CIR)
6.3.4 APB2外设复位寄存器 (RCC_APB2RSTR)
6.3.5 APB1外设复位寄存器 (RCC_APB1RSTR)
6.3.6 AHB外设时钟使能寄存器 (RCC_AHBENR)
6.3.7 APB2外设时钟使能寄存器(RCC_APB2ENR)
6.3.8 APB1外设时钟使能寄存器(RCC_APB1ENR)
6.3.9 备份域控制寄存器 (RCC_BDCR)
6.3.10 控制/状态寄存器 (RCC_CSR)

图 4-30 RCC 文档解释与代码对应章节图

可以看到上图，每个寄存器都是 32 位的，左边是而且顺序刚好分别对应，结构体是会分配内存的，这样这些 C 语言中的 struct 结构体中定义的成员会对应映射到对应的寄存器上，那么我们就可以通过操纵程序中的该结构体的对应成员，就相当于操作的是对应的寄存器，这个是 C 语言和单片机软硬件对应上的又一大关键点，请不熟悉的读者好好理解一下，如实在不理解，可以致电 STM32 神舟系列开发板的官方工程师。

4.4.8 代码剖析 3---main函数寄存器级分析（重点）

1. LED 灯为什么会一亮一灭呢？

```
int main(void)   //main 是程序入口
{
    /*  使能 APB2 总线的时钟，对 GPIO 的端口 B 时钟使能  */
    RCC->APB2ENR |= RCC_APB2Periph_GPIOB;

    /*-- GPIO Mode Configuration 速度，输入或输出  -----------------------*/
    /*-- GPIO CRL Configuration  设置 IO 端口低 8 位的模式（输入还是输出）---*/
    /*-- GPIO CRH Configuration  设置 IO 端口高 8 位的模式（输入还是输出）---*/
    GPIOB->CRH &=   0xFFFFFFF0;
    GPIOB->CRH |=   0x00000003;

    while (1)
```

```
    {
        GPIOB->BRR = GPIO_Pin_8;
        Delay(0x2FFFFF);   //管脚输出低电平，LED 灯亮
        GPIOB->BSRR = GPIO_Pin_8;
        Delay(0x2FFFFF);   //管脚输出高电平，LED 灯灭
    }
}
```

我们看一下 STM32 中文参考手册中的 GPIOx_BRR 和 GPIOx_BSRR 两个寄存器，进入到文档的 77 页，我们以分析 GPIOx_BSRR 为例，GPIOx_BRR 是同样的原理，如下表 4-4、表 4-5 所示：

表 4-4 GPIOx_BSRR 寄存器

BIT 31	BIT 30	BIT 29	BIT 28	BIT 27	BIT 26	BIT 25	BIT 24
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8
w	w	w	w	w	w	w	w

BIT 23	BIT 22	BIT 21	BIT 20	BIT 19	BIT 18	BIT 17	BIT 16
BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w

BIT 15	BIT 14	BIT 13	BIT 12	BIT 11	BIT 10	BIT 9	BIT 8
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8
w	w	w	w	w	w	w	w

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT0
BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w

表 4-5 GPIOx_BSRR 寄存器功能说明

位31:16	**BRy**: 清除端口x的位y (y = 0…15) 这些位只能写入并只能以字(16位)的形式操作。 0：对对应的ODRy位不产生影响 1：清除对应的ODRy位为0 注：如果同时设置了 BSy 和 BRy 的对应位，BSy 位起作用。
位15:0	**BSy**: 设置端口x的位y (y = 0…15) 这些位只能写入并只能以字(16位)的形式操作。 0：对对应的ODRy位不产生影响 1：设置对应的 ODRy 位为 1

分析1：从以上两个寄存器里的内容可以知道到：

- ODRy = 1就会输出高电平，如果是高电平，我们查看了原理图，这个LED灯灭，可以通过操作 GPIOx_BSRR寄存器的对应位来改变

- ODRy = 0就会输出低电平，如果是低电平，我们查看了原理图，这个LED灯亮，可以通过操作 GPIOx_BRR寄存器的对应位来改变

分析2：进一步分析，我们如何通过代码来改变这个ODRy呢？大家请看下面：

- 代码GPIOB->BRR = GPIO_Pin_8;可以使得GPIOx_BRR寄存器的BR8位为1，这样就是的GPIO端口B的对应ODR8=0，即PB8管脚输出低电平，使得DS1亮。
- 代码GPIOB->BSRR = GPIO_Pin_8;可以使得GPIOx_BSRR寄存器的BS8位为1，这样就是的GPIO端口B的对应ODR8=1，即PB8管脚输出高电平，使得DS1灭。

2. 要使用 PB8 管脚需要做哪些初始化的工作呢？如下图 4-31

```
int main(void)    //main是程序入口
{
    /* 使能APB2总线的时钟，对GPIO的端口B时钟
    RCC->APB2ENR |= RCC_APB2Periph_GPIOB;

    /*-- GPIO Mode Configuration速度，输入或输出 -----------------------*/
    /*-- GPIO CRL Configuration 设置IO端口低8位的模式（输入还是输出）---*/
    /*-- GPIO CRH Configuration 设置IO端口高8位的模式（输入还是输出）---*/
    GPIOB->CRH &=  0xFFFFFFF0;
    GPIOB->CRH |=  0x00000003;

    while (1)
    {
        GPIOB->BRR = GPIO_Pin_8;
        Delay(0x2FFFFF);
        GPIOB->BSRR = GPIO_Pin_8;
        Delay(0x2FFFFF);
    }
}
```

初始化GPIOB端口的时钟

配置GPIOB端口的模式

图 4-31 PB8 初始化

分析1：使能APB2总线上的GPIO端口B的时钟，我们可以看下系统图，如下图4-32，可以看到GPIOB是属于APB2总线管理的，那么如何初始化这个是ST公司要求的，而不是我们STM32神舟系列开发板官方规定的，一切都是STM32芯片厂家ST公司制定的。

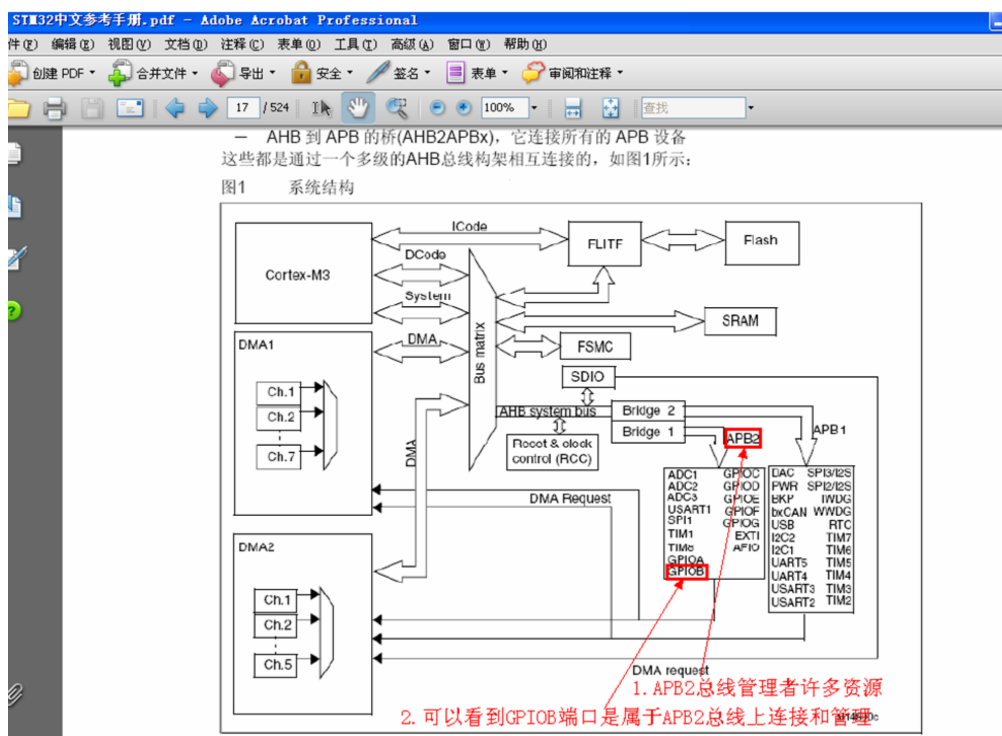


图 4-32 STM32 系统图

分析 2：代码 RCC->APB2ENR |= RCC_APB2Periph_GPIOB;是使能 GPIOB 的时钟，我们找到寄存器 RCC_APB2ENR 如图 4-33，仔细看看是什么操作的

图 4-33 寄存器 RCC_APB2ENR

点击进入文档第 61 页可以看到 6.3.7 节，可以看到 GPIOB 端口的时钟设置选项，如图 4-34：

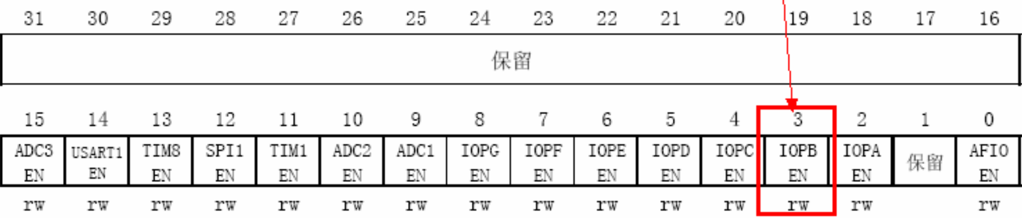
6.3.7 APB2 外设时钟使能寄存器(RCC_APB2ENR)

偏移地址：0x18

复位值：0x0000 0000

访问：字，半字和字节访问

通常无访问等待周期。但在APB2总线上的外设被访问时，将插入等待状态直到外设访问结束。



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADC3 EN	USART1 EN	TIM8 EN	SPI1 EN	TIM1 EN	ADC2 EN	ADC1 EN	IOPG EN	IOPF EN	IOPE EN	IOPD EN	IOPC EN	IOPB EN	IOPA EN	保留	AFIO EN
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw		rw

图 4-34 GPIOB 端口的时钟设置

看下第三位的使能时钟位，如图 4-35：

位5	**IOPDEN**：IO端口D时钟使能 由软件置'1'或清'0' 0：IO端口D时钟关闭； 1：IO端口D时钟开启。
位4	**IOPCEN**：IO端口C时钟使能 由软件置'1'或清'0' 0：IO端口C时钟关闭； 1：IO端口C时钟开启。
位3	**IOPBEN**：IO端口B时钟使能 由软件置'1'或清'0' 0：IO端口B时钟关闭； 1：IO端口B时钟开启。 设置这位为1，表示GPIO端口B的时钟开启
位2	**IOPAEN**：IO端口A时钟使能 由软件置'1'或清'0' 0：IO端口A时钟关闭； 1：IO端口A时钟开启。
位1	保留，始终读为0。
位0	**AFIOEN**：辅助功能IO时钟使能 由软件置'1'或清'0' 0：辅助功能IO时钟关闭； 1：辅助功能IO时钟开启。

图 4-35 时钟使能位

我们通过代码 RCC->APB2ENR |= RCC_APB2Periph_GPIOB;来对 RCC_APB2ENR 寄存器的位 3 进行操作置位，那么 RCC_APB2Periph_GPIOB 的值请见：

代码：#define RCC_APB2Periph_GPIOB ((uint32_t)0x00000008)

从这句代码可以知道 0x00000008 的 8 化成二进制是 1000，刚好是对 RCC->APB2ENR 寄存器也就是 RCC_APB2ENR 寄存器的第 3 位置位，使得 IOPB EN 为 1，使得 IO 端口 B 时钟使能。

分析 3：配置 GPIO 端口 B 的工作模式，我们这里可以看到它配置改变了 CRH 这个寄存器。

```
/*-- GPIO Mode Configuration 速度，输入或输出 -----------------------*/
/*-- GPIO CRL Configuration 设置 IO 端口低 8 位的模式（输入还是输出）---*/
/*-- GPIO CRH Configuration 设置 IO 端口高 8 位的模式（输入还是输出）---*/
GPIOB->CRH &=  0xFFFFFFF0;
GPIOB->CRH |=  0x00000003;
```

我们找到这个 CRH 寄存器完整的名称叫 GPIOx_CRH 寄存器的内容，偏移地址：0x04，复位值：0x4444 4444，如下表 4-6 所示：

表 4-6 GPIOx_CRH 寄存器

31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16
CNF15[1：0]	MODE15[1：0]	CNF14[1：0]	MODE14[1：0]	CNF13[1：0]	MODE13[1：0]	CNF12[1：0]	MODE12[1：0]
15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
CNF11[1：0]	MODE11[1：0]	CNF10[1：0]	MODE10[1：0]	CNF9[1：0]	MODE9[1：0]	CNF8[1：0]	MODE8[1：0]

我们开始分析一下代码：

- GPIOB->CRH &= 0xFFFFFFF0;

GPIOB->CRh &= 0xFFFFF0FF = 1111 1111 1111 1111 1111 1111 1111 0000

可以看到将 CRH 寄存器的第 0、1、2、3 位清 0，其他位默认值不变，看上表可以知道，这 4 位刚好是管脚 PB8 的配置寄存器 如下表 4- 7 所示：

表 4- 7 PB8 的配置寄存器

3　　　　2	1　　　　0
CNF8[1：0]	MODE8[1：0]

- GPIOB->CRH |= 0x00000003;

GPIOB->CRH |= 0x00000003 = 0000 0000 0000 0000 0000 0000 0000 0011

下表 4-8 可以看到将 CRH 寄存器的第 0、1、2、3 位

表 4-8 CRH 寄存器

3　　　　2	1　　　　0
CNF8[1：0]	MODE8[1：0]

分别置成 1、1、0、0，其他位的 GPIO 端口值都清 0，意思就是只配置 PB8 这个管脚，其他 PB 口的管脚配置寄存器都全部变成 0，这样 MODE8=11；CNF8=00；查表可以知道，如下图 4-36：

位	说明
位31:30 27:26 23:22 19:18 15:14 11:10 7:6 3:2	**CNFy[1:0]**：端口x配置位(y = 8…15) 软件通过这些位配置相应的I/O端口，请参考表15端口位配置表。 在输入模式(MODE[1:0]=00)： 00：模拟输入模式 01：浮空输入模式(复位后的状态) 10：上拉/下拉输入模式 11：保留 在输出模式(MODE[1:0]>00)： 00：通用推挽输出模式 01：通用开漏输出模式 10：复用功能推挽输出模式 11：复用功能开漏输出模式
位9:28 25:24 21:20 17:16 13:12 9:8, 5:4 1:0	**MODEy[1:0]**：端口x的模式位(y = 8…15) 软件通过这些位配置相应的I/O端口，请参考表15端口位配置表。 00：输入模式(复位后的状态) 01：输出模式，最大速度10MHz 10：输出模式，最大速度2MHz 11：输出模式，最大速度50MHz

2. 因为MODE8大于0，所以CNF8的配置应该查看表的这一栏目

3. CNF8=00是通用推挽输出模式

1. MODE8=11是50MHZ速度的输出模式

图 4-36 配置 GPIO 管脚工作状态

这样配置后 PB8 被配置成输出模式，输出速度为 50MHz，状态是通用推挽输出模式；因为我们这个例程中需要点 LED 灯，这是一种输出模式。

4.4.9 代码下载方式 1---通过J-Flash下载

这里详细介绍通过JLINK 仿真器下载固件到ARM核心板上的过程。

JLINK

V8是目前主流的JTAG仿真器，支持所有的ARM7/9/11和Cortex-M0/M1/M3处理器。而且与主流的开发环境，如神舟系列STM32开发板采用的IAR,MDK开发环境完美的结合。通过JLINK仿真器，我们可以方便的下载，和在线调试代码。因此，推荐ARM核心板与JLINK V8搭配使用。

以下详细描述使用JLINK V8下载固件到ARM核心板的过程。

第一步，按下图4-40连接好JLINK V8, ARM核心板与PC机。ARM核心板上电。

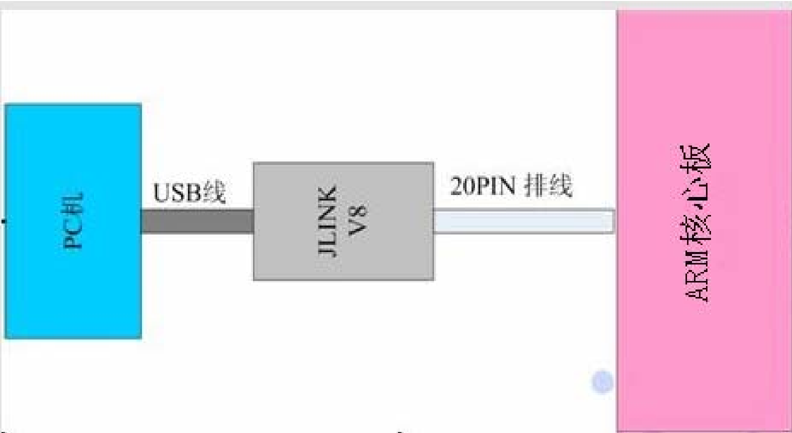


图 4-37 连接好 JLINK V8 仿真器与 ARM 核心板

第二步，在PC机上打开J-Flash ARM软件，在开始菜单中找到“SEGGER”文件夹（前提要先安装了仿真器的驱动），如下图4-38：.

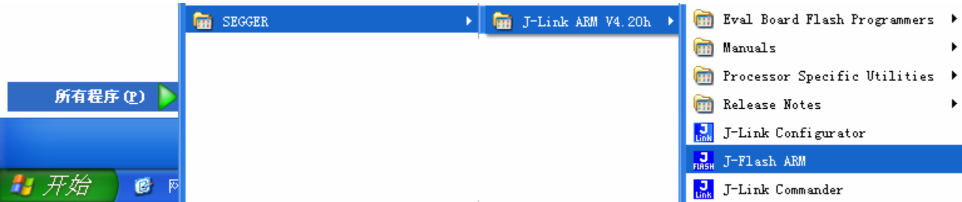


图 4-38 打开仿真器 J-Flash 工具

运行 J-Flash ARM，界面如下图 4-39

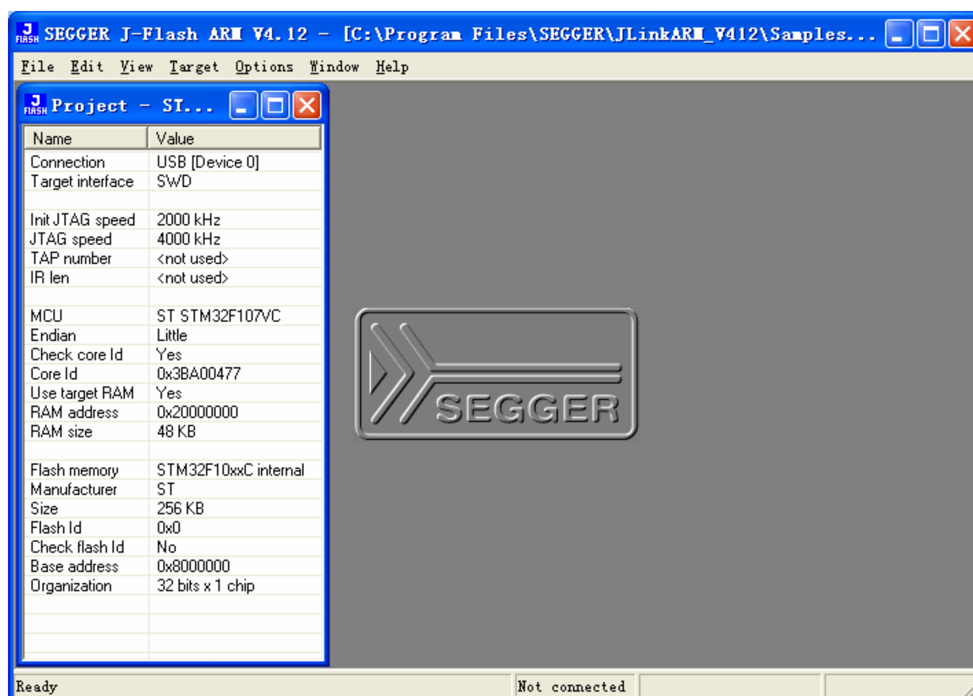


图 4-39 J-Flash 页面

然后通过“File”菜单下的“Open…”来打开需要烧写的文件，可以是.bin 格式，也可以是.hex 格式，甚至可以是.mot 格式。注意起始地址如图 4-40、4-41。

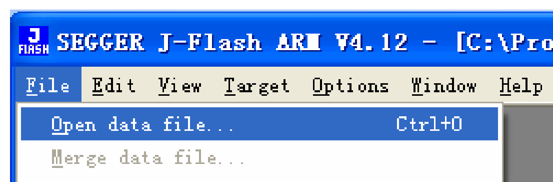


图 4-40 打开程序文件

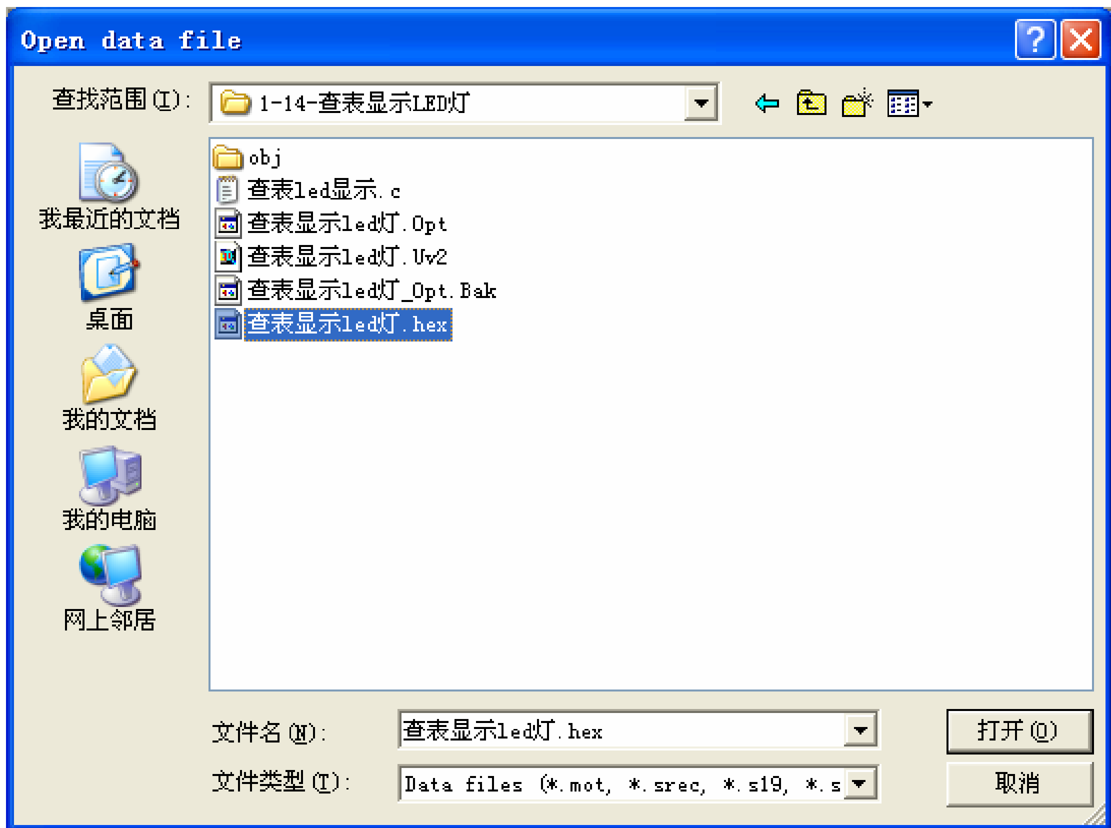


图 4-41 打开程序文件

首次使用的时候应该在 File 菜单，选择 Open Project ，选择你的目标芯片，如下图 4-42：

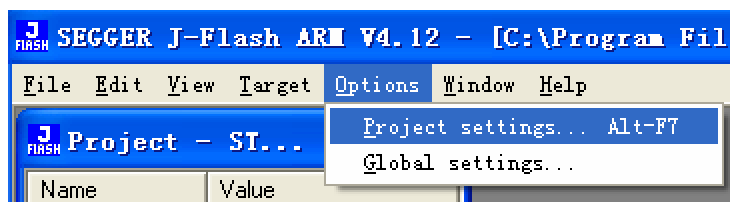


图 4-42 打开配置页面

接下来在“Options”选择“Project settings”在弹出的对话框中，点击 CPU 标签，选择神舟 51ARM 核心板的处理器 STM32F103C8T6，如下图 4-43。

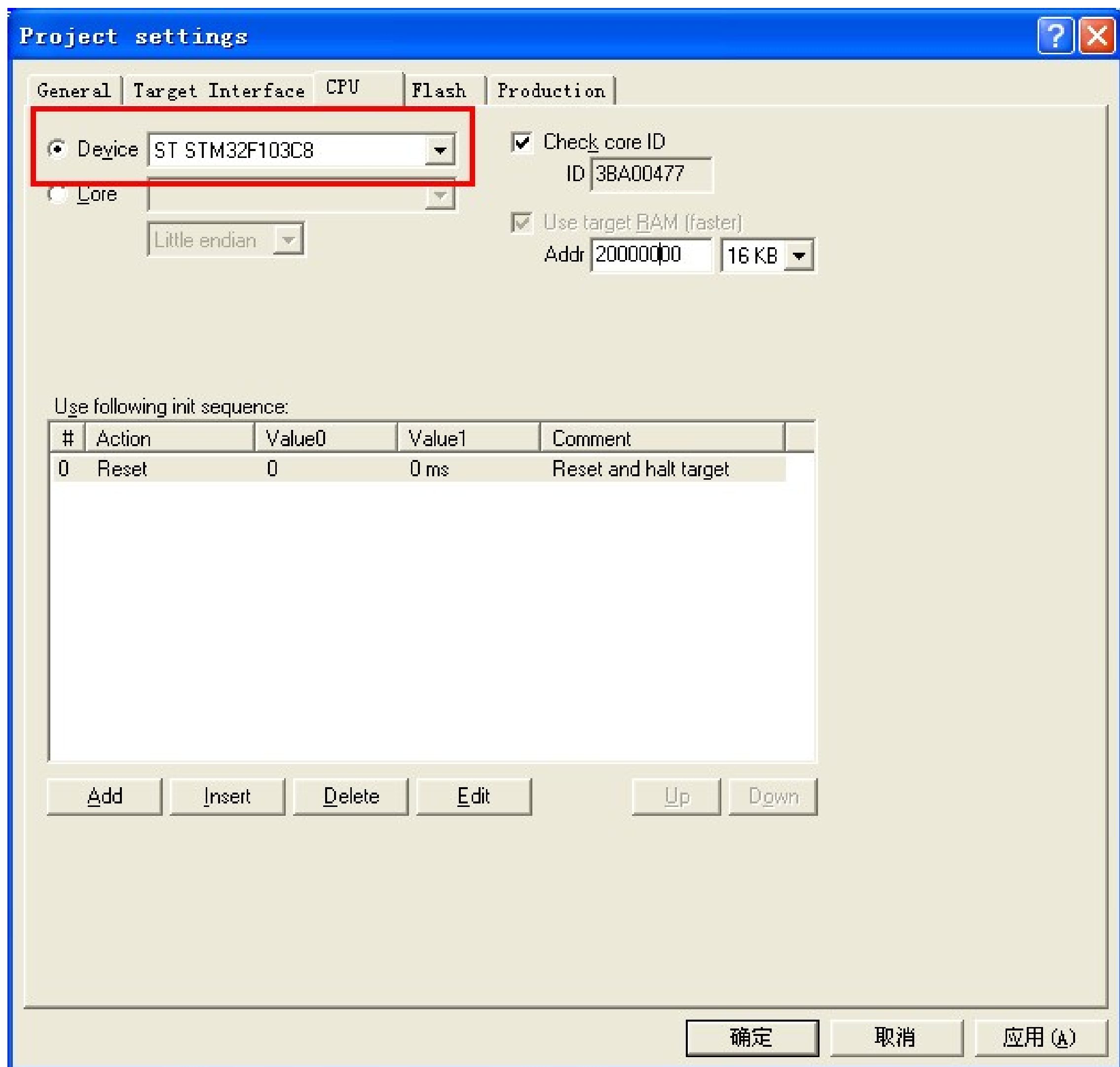


图 4-43 选择 ARM 处理器

点击 Target Interface 选择使用 JTAG 还是 SWD 接口，JLINK V8 可以选择用 JTAG 或 SWD 接口来下载，在线调试，而这两种接口 STM32 处理器也都是支持的。如下图 4-44 所示：

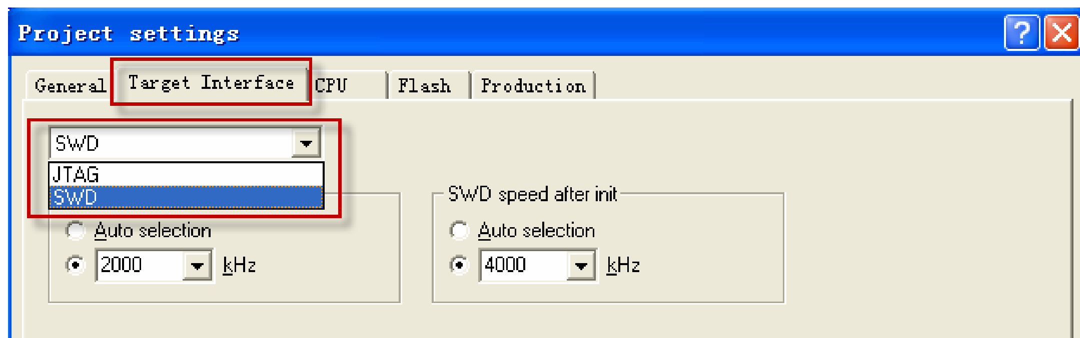


图 4-44 选择下载方式

设置好之后，就可以到 Target 里面进行操作，一般步骤是先“Connect”，然后“Erase Chip”，然后“Program”。大部分芯片还可以加密，主要的操作都在 Target 菜单下完成。为了方便操作，我们可以直接按 F7 快捷键，自动擦除和烧录固件。如下图 4-45 所示：

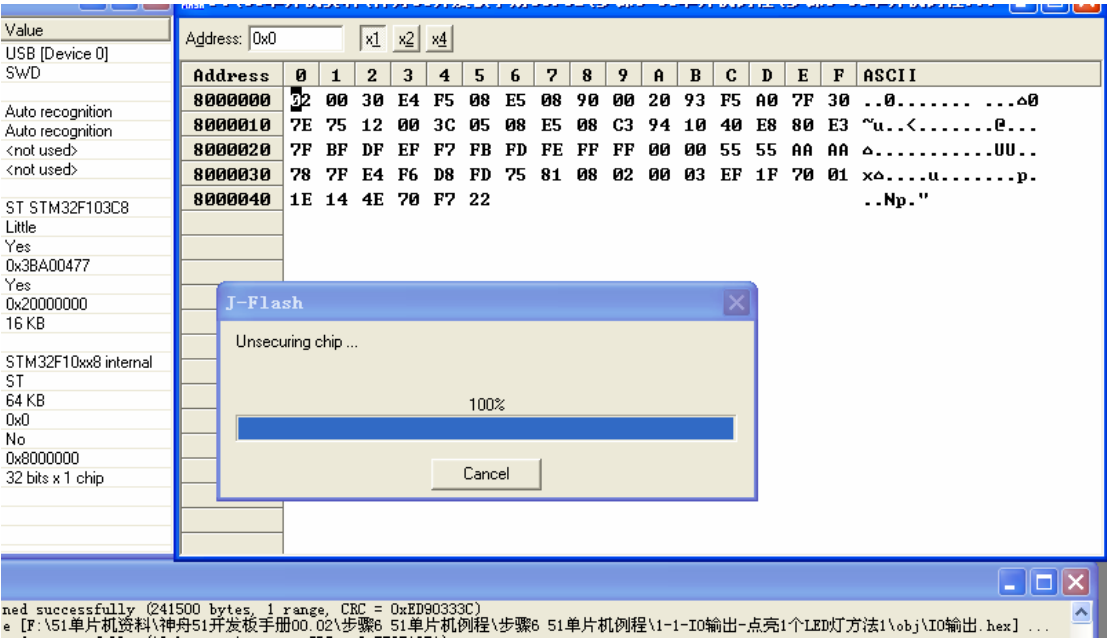


图 4-45 程序的下载

烧录完成，如下图4-46

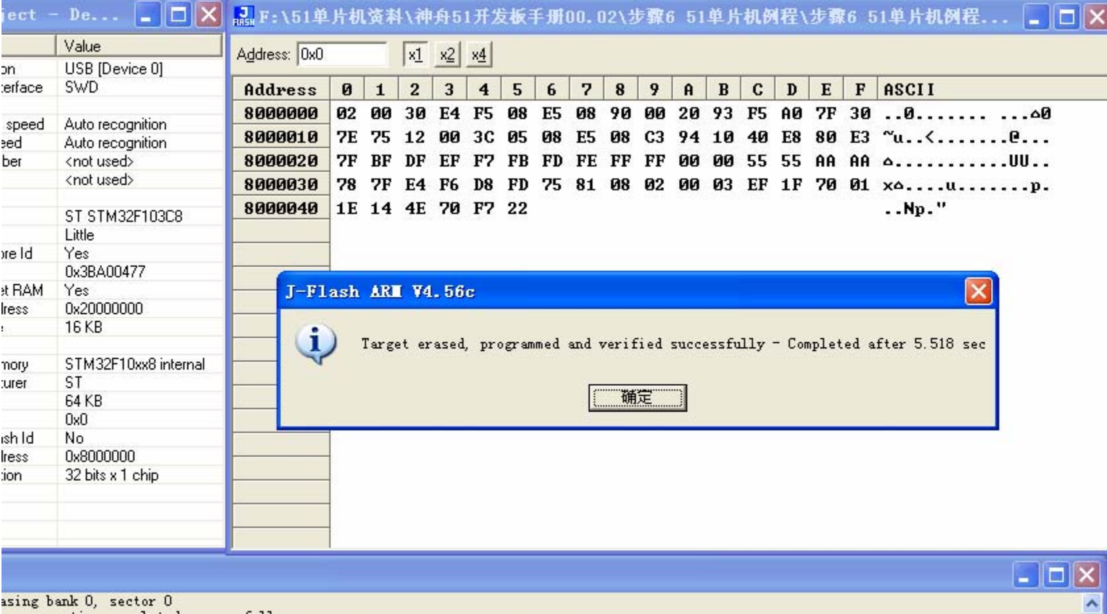


图 4-46 程序下载完成

我们通过MDK编译出来的Hex文件的程序也可以通过该方法下载到核心板上
关于J-FLASH ARM更详细的操作请参阅JLINK的用户手册。

4.4.10 代码下载方式 2---通过KEIL软件直接下载

打开工程在代码编译通过后，我们需要验证程序是否与我们的设计预期相符合，一般有几种方式来验证。

第一种，软件仿真。在没有硬件环境的情况下，我们可以使用 MDK 的软件仿真功能来对代码的功能进行初步验证。关于软件仿真功能的使用，在本文档中不在详细描述，有兴趣的朋友，可以常看 MDK 相关手册和说明文档。

第二种，在线仿真。如果硬件环境允许的情况下，建议使用在线仿真来进行调试，完全与实际应用相符。开发板所有实验程序都使用 JLINK 仿真器仿真调试通过。

JLINK 是一款主流的支持 ARM 内核芯片的 JTAG 仿真器，配合 IAR，KEIL，WINARM，RealView 等集成开发环境，支持所有 ARM7/ARM9/Cortex-M3 内核芯片的仿真。在这里，我们以 ARM 核心板的入门程序为例，说明如何在 MDK 开发环境中，搭配 JLINK 仿真器，在线仿真调试程序。

1）按下图，连接JLINK仿真器与ARM核心板。并给ARM核心板上电，如下图4-47。

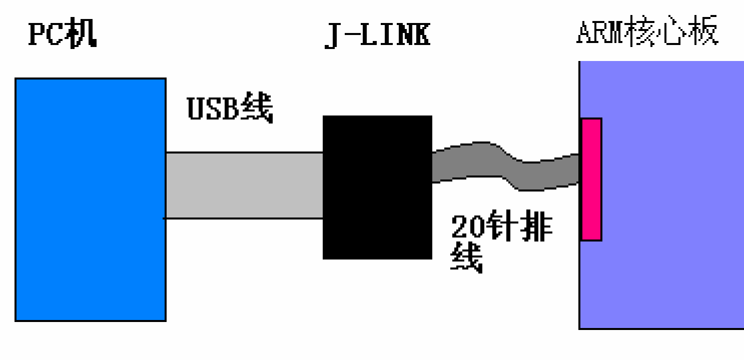


图 4-47 硬件环境连接

这个工程默认的是软件仿真，如果开发板要用J-LINK
调试的话，还需要在开发环境中做如下修改。实际上，我们开发程序的时候80%都是在硬件上调试的。

2）具体配置如下图4-54所示：点击，在Debug选项里如下图4-49：

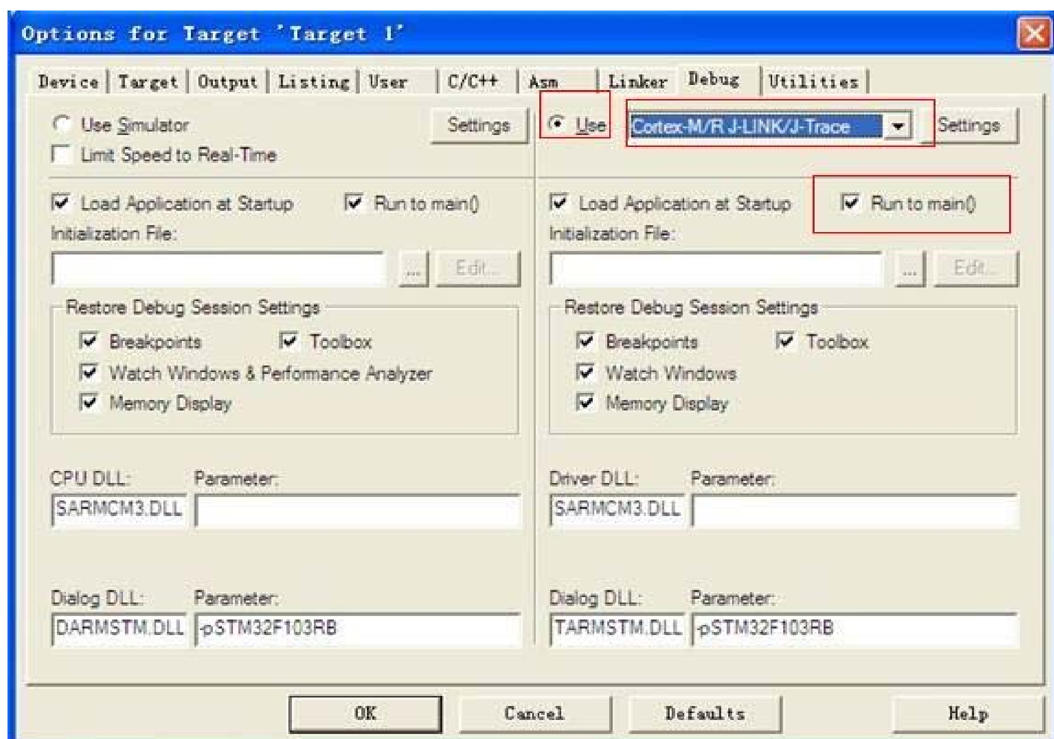


图 4-49　KEI 软件的 Debug 设置

3）点Settings按钮，查看是否识别到了目标板CPU（注意，此处目标板应该上电，并将JLINK V8与模板连接好，JLINK V8也需要上电）如下图4-50所示

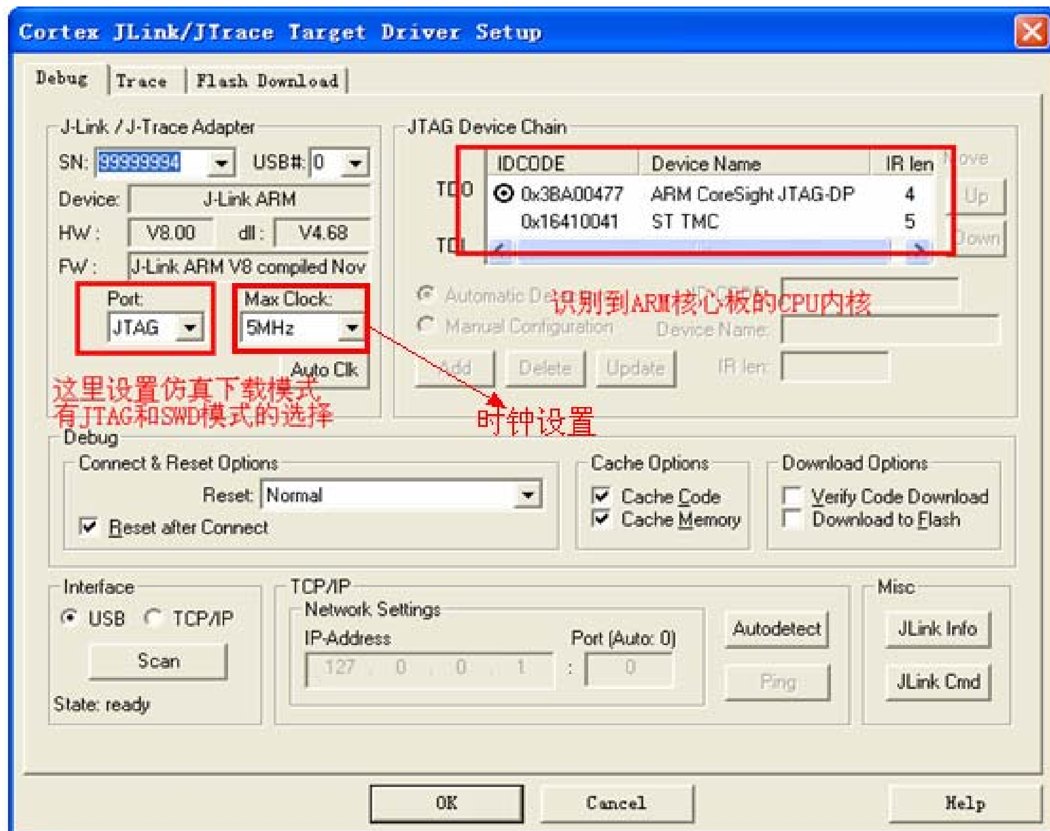


图 4-50 查看连接情况

设置完点确定

4）下面选择Utilities选项卡，如下图4-51所示

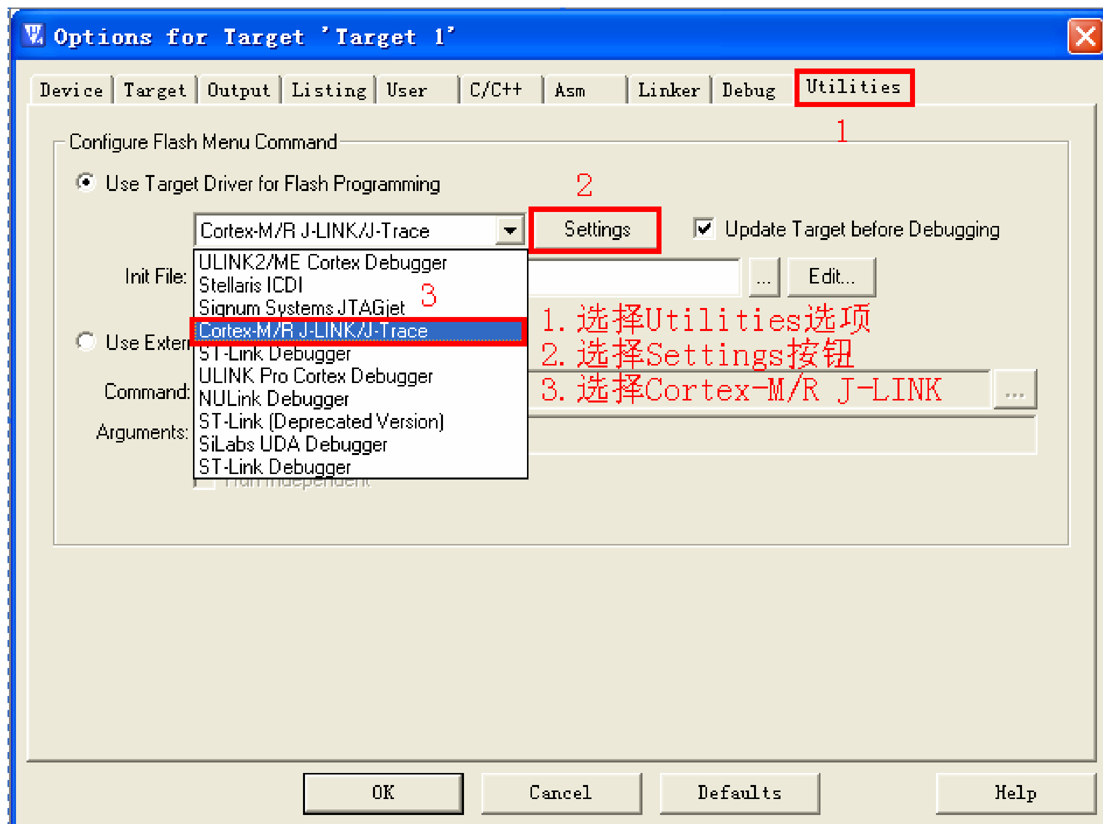


图 4-51 Utilities 选项卡设置

5）在选项卡Utilities\Setting\Flash download中设置我们的Flash，如下图4-52、4-53和4-54，因为103C8T6不足128K的Flash，我们选128K就好：

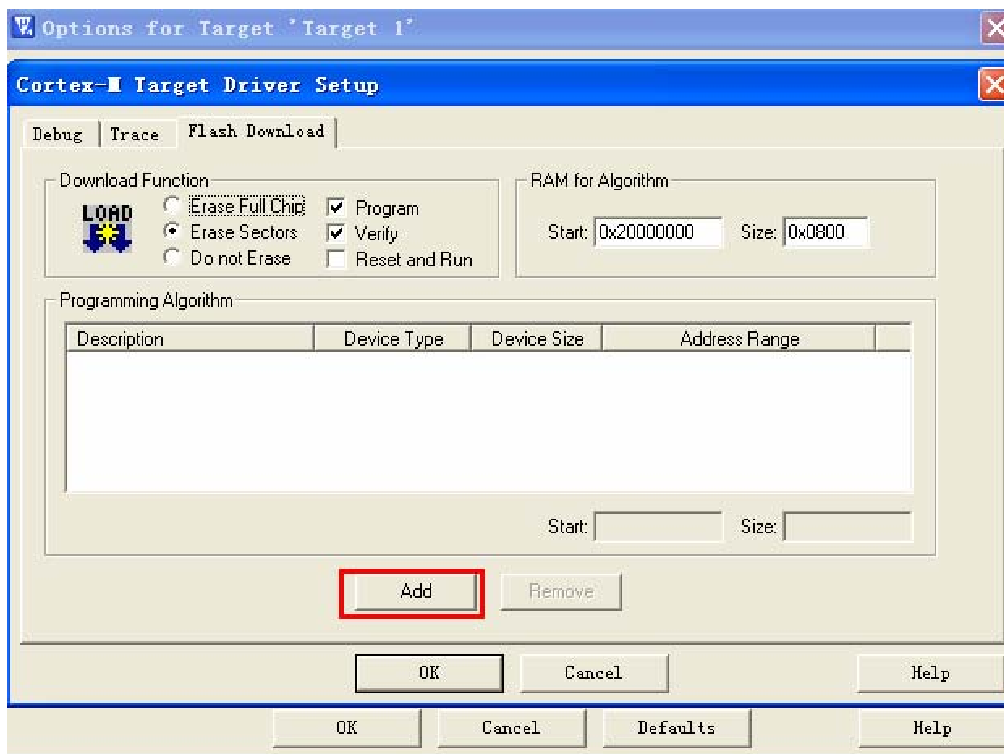


图 4-52 Flash 的配置

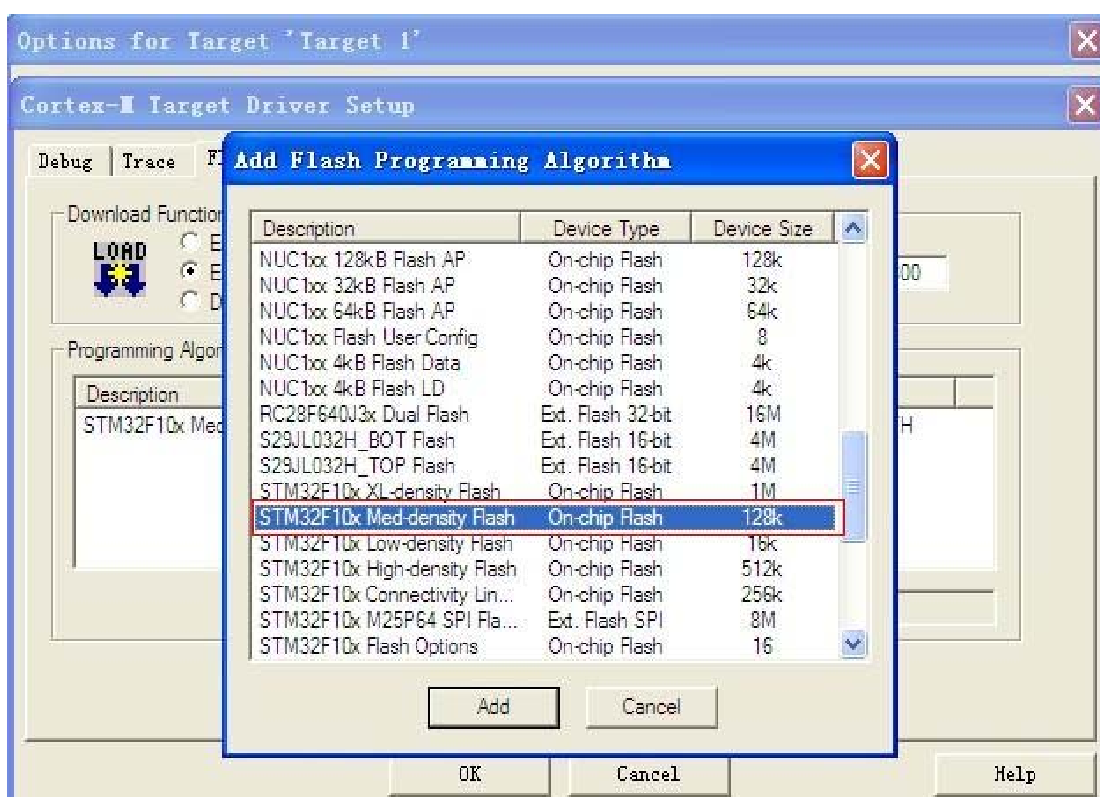


图 4-53 选择 Flash 的大小

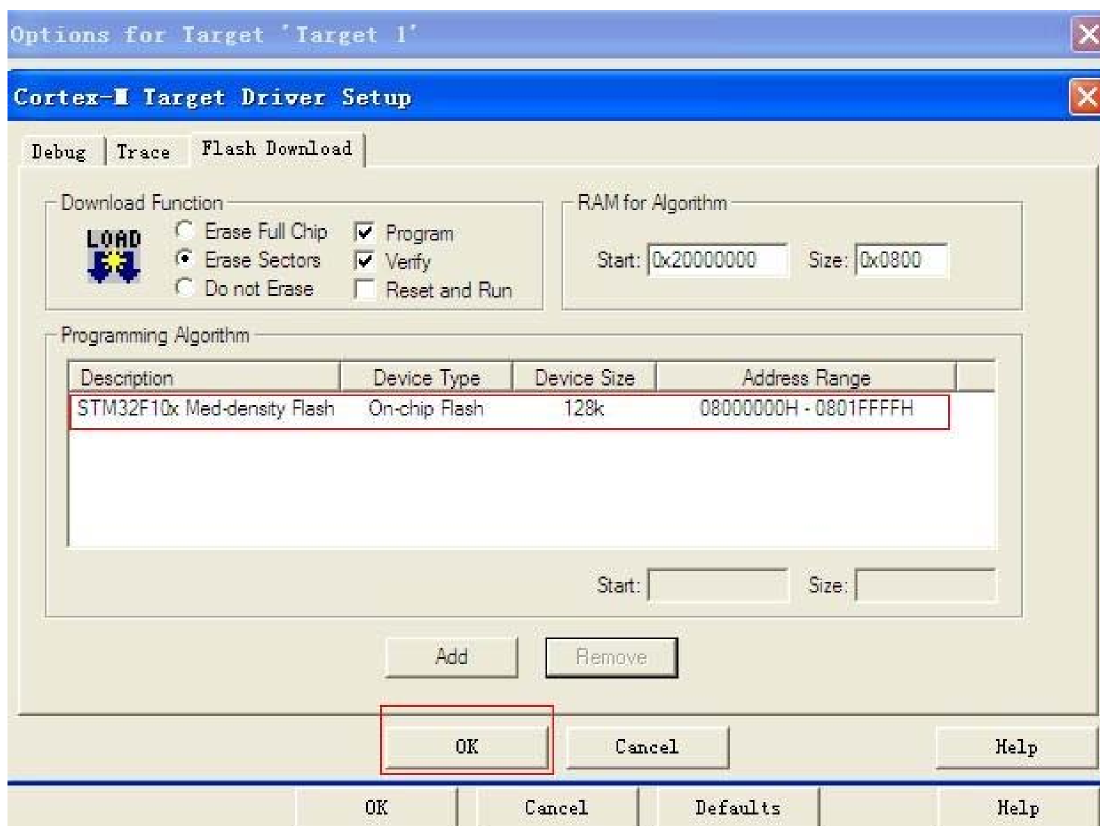


图 4-54 配置 Flash 完成

设置STM32F103C8T6芯片的内部ROM大小是128K点击OK按钮

6）编译全部代码，然后点 LOAD 按钮下载程序到目标板，如图4-55

```
Build Output
compiling main.c...
linking...
Program Size: Code=412 RO-data=252 RW-data=0 ZI-data=608
FromELF: creating hex file...
"STM32-DEMO.axf" - 0 Error(s), 0 Warning(s).
```

图 4-55 下载程序到 ARM 核心板上

到了这里就算是大功告成了。如果在新建工程中遇到什么问题，先不要急，可先参考STM32神舟系列的其他相关资料。

在完成以上操作以后，一个MDK工程就设置完成了，只需要依据实际应用要求编写相关的代码即可。

4.5 从零开始搭建一个最简单的模版

4.5.1 如何去官网下载最新的STM32 资料

1、 打开网页，在百度搜索网页栏中搜索“ST”.

2、在搜索页面点击 st 意法半导体进入 ST 官网 。

3、在 ST 官网上点击“微控制器”如下图 4-56 所示：



图 4-56 选择“微控制器”

4、选择 STM32-32 位的微控制器，如下图 4-57：



图 4-57 选择 STM32 32 位的 ARM

5、选择 F1 系列，因为我们的 ARM 核心板的处理器是 F1 系列的，如图 4-58 所示：



图 4-58 点击 F1 系列

6、选择你需要的芯片，比如 STM32F103ZET6、STM32F107VCT6，STM32F207ZGT、STM32F407ZGT 等，这里我们用 STM32F103C8T6 做示范，如下图 4-59 所示：

STM32F102C8	LQFP 48 7x7x1.4	Active	ARM Cortex-M3	48	64	-	10	3	-	2 x
STM32F102CB	LQFP 48 7x7x1.4	Active	ARM Cortex-M3	48	128	-	16	3	-	2 x
STM32F102R4	LQFP 64 10x10x1.4	Active	ARM Cortex-M3	48	16	-	4	2	-	2 x
STM32F102R6	LQFP 64 10x10x1.4	Active	ARM Cortex-M3	48	32	-	6	2	-	2 x
STM32F102R8	LQFP 64 10x10x1.4	Active	ARM Cortex-M3	48	64	-	10	3	-	2 x
STM32F102RB	LQFP 64 10x10x1.4	Active	ARM Cortex-M3	48	128	-	16	3	-	2 x
STM32F103C4	LQFP 48 7x7x1.4	Active	ARM Cortex-M3	72	16	-	6	3	-	2 x
STM32F103C6	LQFP 48 7x7x1.4; UFQFPN...	Active	ARM Cortex-M3	72	32	-	10	3	-	2 x
STM32F103C8	LQFP 48 7x7x1.4	Active	ARM Cortex-M3	72	64	-	20	4	-	2
STM32F103CB	LQFP 48 7x7x1.4; UFQFPN...	Active	ARM Cortex-M3	72	128	-	20	4	-	2 x
STM32F103R4	[illegible]0x10x1.4; TFBG...	Active	ARM Cortex-M3	72	16	-	6	3	-	2 x
STM32F103R6	[illegible]10x10x1.4; TFBG...	Active	ARM Cortex-M3	72	32	-	10	3	-	2 x
STM32F103R8	LQFP 64 10x10x1.4; TFBG...	Active	ARM Cortex-M3	72	64	-	20	4	-	2 x
STM32F103RB	LQFP 64 10x10x1.4; TFBG...	Active	ARM Cortex-M3	72	128	-	20	4	-	2 x
STM32F103RC	LQFP 64 10x10x1.4	Active	ARM Cortex-M3	72	256	-	48	8	-	2 x
STM32F103RD	LQFP 64 10x10x1.4; WLC...	Active	ARM Cortex-M3	72	384	-	64	8	-	2 x
STM32F103RE	LQFP 64 10x10x1.4; WLC...	Active	ARM Cortex-M3	72	512	-	64	8	-	2 x

图 4-59 找到我们的“STM32F103C8”处理器芯片

7、点击对应的芯片后，在跳转的页面中，点击“所有”，将网页往下拉，找到官网提供的文档资料如图 4-60：



图 4-60 点击“所有”选项

8、往下拉，找到它的标准外设库，如图 4-61 所示：

STSW-STM32027	Communication peripheral FIFO emulation with DMA and DMA timeout in STM32F10x microcontrollers (AN3109)
STSW-STM32028	STM32's ADC modes and their applications (AN3116)
STSW-STM32033	STM32F1xx motor control firmware library for the L6470 dSPIN IC
STSW-STM32047	Implementing receivers for infrared remote control protocols using STM32F1 microcontrollers (AN3174)
STSW-STM32054	STM32F10x standard peripheral library
STSW-STM32056	STM32F1xx motor control firmware for easySPIN L6474
STSW-STM32080	DfuSe USB device firmware upgrade STMicroelectronics extension: contains the demo GUI, debugging GUI, all sources files and the protocol layer (UM0412)
STSW-STM32086	CEC (consumer electronic control) C library using the STM32F101xx, STM32F102xx and STM32F103xx microcontrollers (UM0685)
STSW-STM32093	STM32 TFT-LCD direct drive demonstration firmware (AN3241)
STSW-STM32094	STM32 in-application programming over the I2C bus (AN3078)
STSW-STM32098	STM32 embedded GUI library (AN3128)

图 4-61 找到我们的外围库文件

9、下载我们找到的外设库，具体请大家登陆官网查看，比如我要下载一个最新的库，大家看下图，这个库文件我找到了，提示说是 3.5.0 版本的库文件，如下图 4-62 所示：



图 4-62 下载我们的库文件

4.5.2 获取ST库源码

在新建工程模板之前，我们首先需要获取到ST库的源码，源码从上面的流程就可以从ST的官方网站下载到，在这里我们以V3.5.0的库来新建我们的工程模板。

可以看到该库的版本为 3.5.0 版本，下载后如图 4-63 所示：

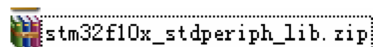


图 4-63 库文件

解压缩之后，真正的标准库函数就在 Libraries 文件夹中，如下图 4-64 所示：

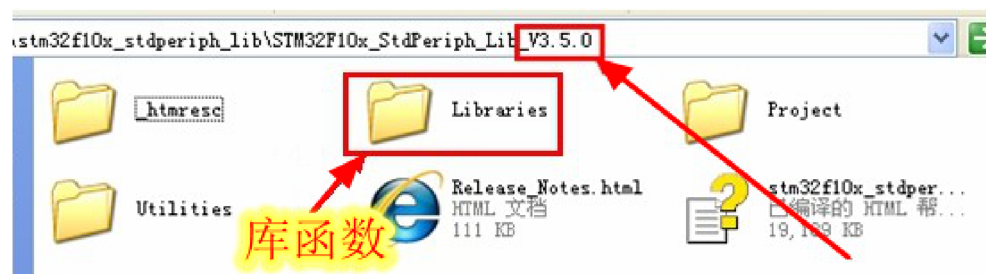


图 4-64 解压我们下载的库文件压缩包

4.5.3 开始新建工程

1. 点击桌面UVision4图标，启动软件，如下图4-65。如果是第一次使用的话会打开一个自带的工程文件，我们可以通过工具栏Project->Close Project选项把它关掉。

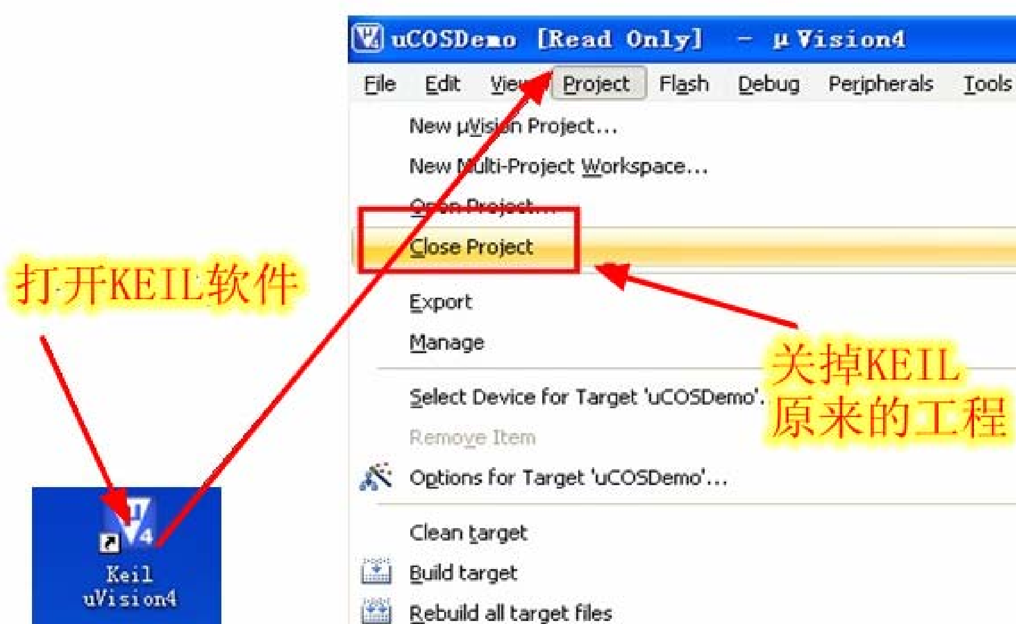


图 4-65 打开 KEIL 软件并且关掉自带的工程

2. 在工具栏Project->New µVision Project...新建我们的工程文件，如下图4-66所示，我们

将新建的工程文件保存在桌面的“STM32神舟开发板模板工程”（先在电脑桌面上新建一个“STM32神舟开发板模板工程”，在该文件夹里面新建一个Project文件夹），文件名取为神舟STM32-DEMO（英文DEMO的意思是例子），名字可以随取，点击保存。如图4-67所示：

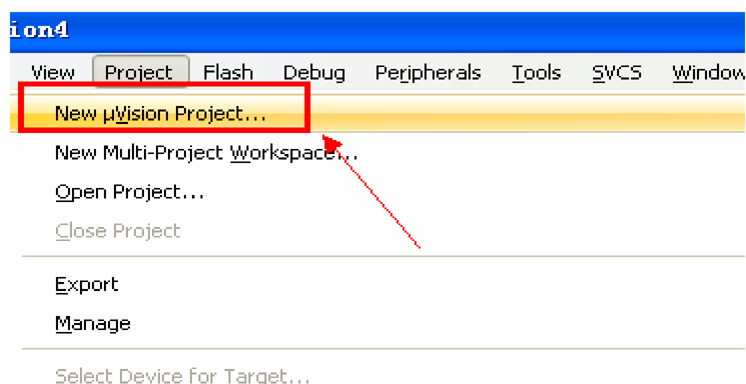


图 4-66 点击“New µVision Project”新建工程

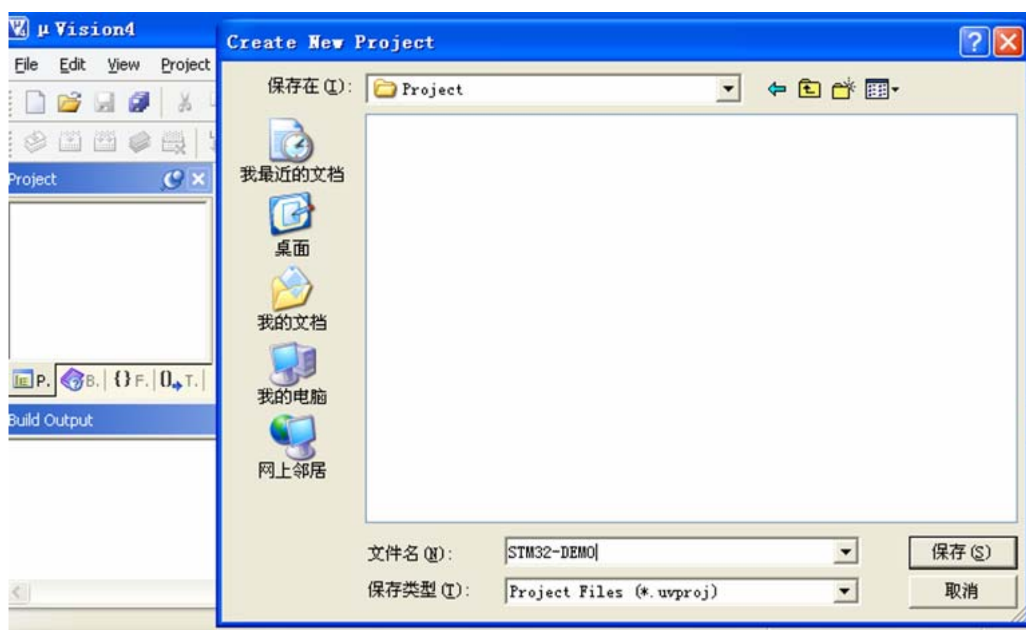


图 4-67 保存我们的新建工程文件

3.接下来的窗口是让我们选择公司跟芯片的型号，我们用神舟 STM32F103C8T6 的板子做举例说明，因为我们的神舟 STM32F103C8T6 的板子用的芯片是 ST 公司的 STM32F103C8T6，有 20K SRAM，128K Flash，属于高集成度的芯片。按如下选择即可，如下图 4-68 所示，选择“ST”公司的芯片，再找到我们的芯片型号即可，如图 4-69 所示：

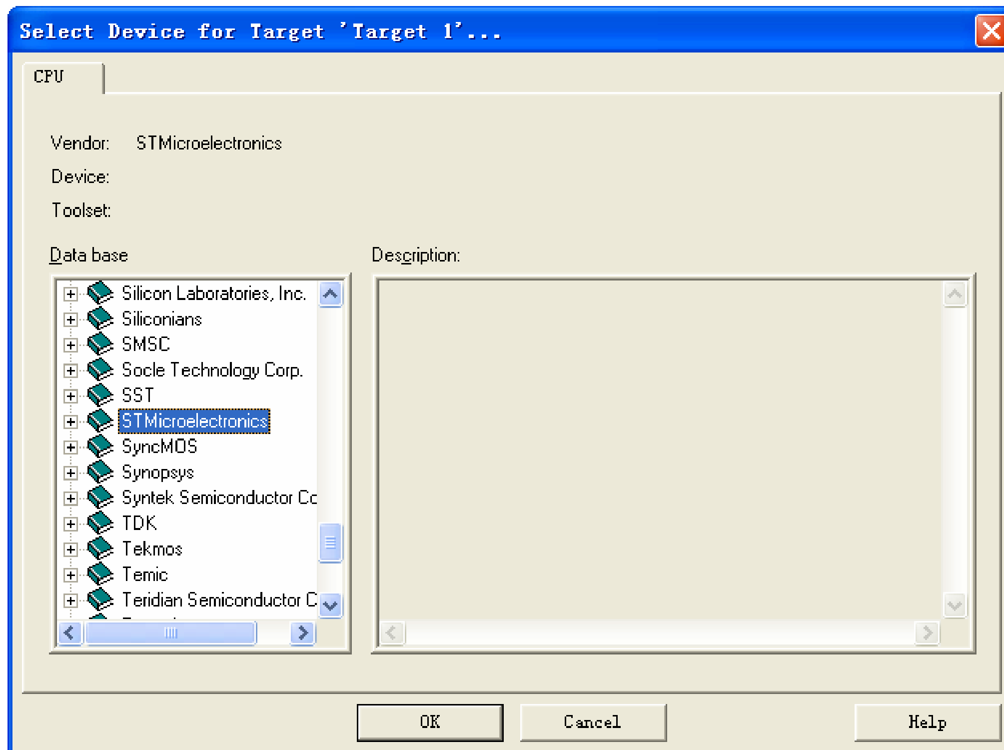


图 4-68 选择芯片公司名称

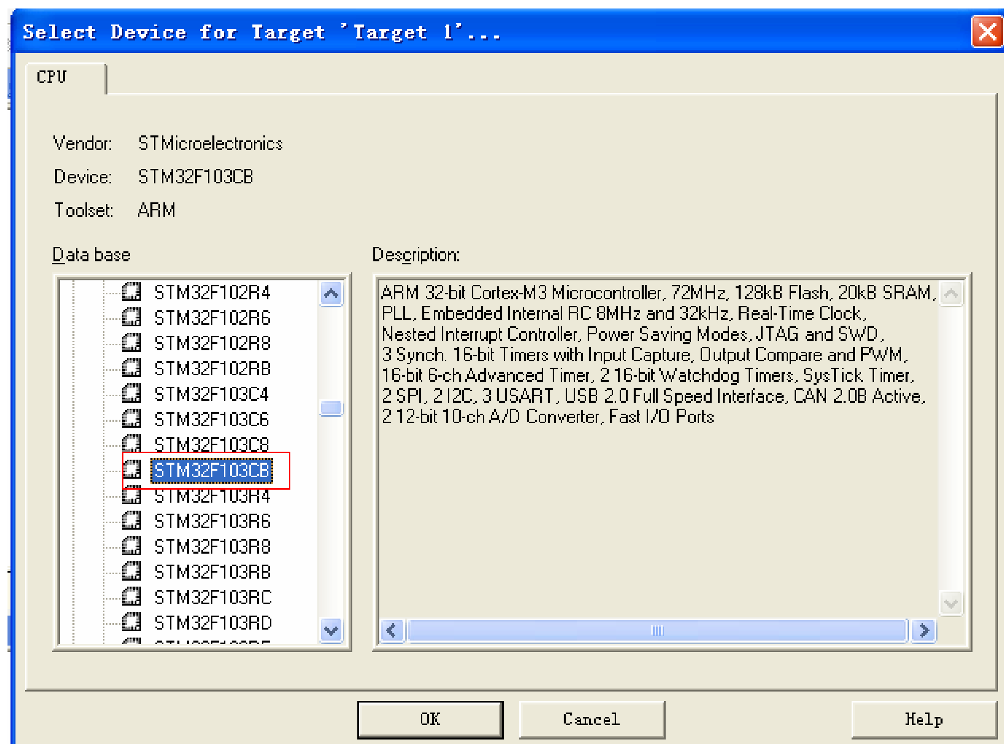


图 4-69 找到我们的芯片型号

4.接下来的窗口问我们是否需要拷贝STM32的启动代码到工程文件中，这份启动代码在M3系列中都是适用的，一般情况下我们都点击是，但我们这里用的是ST的库，库文件里面也自带了这一份启动代码，所以为了保持库的完整性，我们就不需要开发环境为我们自带的启动代码了，稍后我们自己手动添加，这里我们点击否。如图4-70所示：

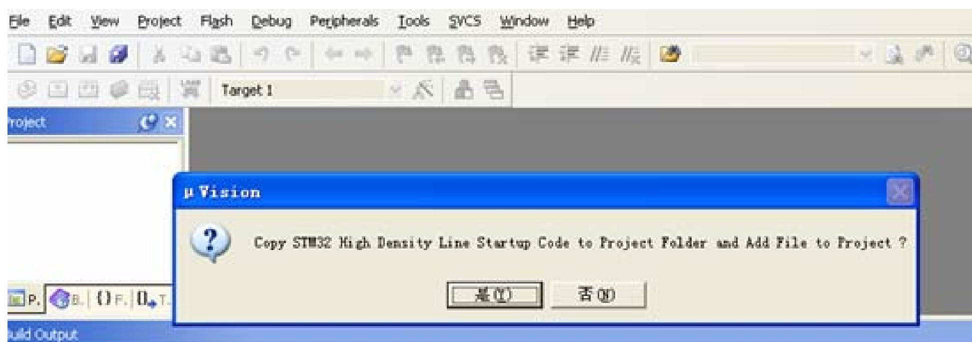


图 4-70 询问是否需要添加自带的启动文件

5. 此时我们的工程新建成功，如下图4-71所示。但我们的工程中还没有任何文件，接下来我们需要在我们的工程中添加所需文件。



图 4-71 新建工程完成后的文件

6. 在“STM32神舟开发板模板工程”文件夹下，我们新建3个文件夹和3个文件，分别为Libraries、Output、Project文件夹以及“删除MDK产生的过程文件.bat”文件、“readme.txt”文件和“stm32f10x_stdperiph_lib.zip”文件；原来新建的Project文件夹用来存放工程文件和用户代码，包括主函数main.c；Libraries用来存放STM32标准库里面的CMSIS和STM32F10x_StdPeriph_Driver这两个文件夹。如下图4-72所示：



图 4-72 新建我们需要用到的文件夹与文件

7. 把从ST官网下载的stm32f10x_stdperiph_lib.zip库函数文件压缩包，解压缩后将\stm32f10x_stdperiph_lib\STM32F10x_StdPeriph_Lib_V3.5.0\Libraries的CMSIS跟STM32F10x_StdPeriph_Driver这两个文件夹拷贝到STM32神舟开发板模板工程\Libraries文件夹中。如下图4-73所示



图 4-73 从库文件中添加需要的文件到我们新建的文件夹中

8. 把标准库目录下的：
stm32f10x_stdperiph_lib\STM32F10x_StdPeriph_Lib_V3.5.0\Project\STM32F10x_StdPeriph_Template文件夹下的main.c、stm32f10x_conf.h、stm32f10x_it.h、 stm32f10x_it.c 拷贝到STM32神舟开发板模板工程\Project目录下。 stm32f10x_it.h、和stm32f10x_it.c这两个文件里面是中断函数，里面为空，并没有写任何的中断服务程序；stm32f10x_conf.h是用户需要配置的头文件，当我们需要用到芯片中的某部分外设的驱动时，我们只需要在该文件下将该驱动的头文件包含进来即可，片上外设的驱动在
Libraries\STM32F10x_StdPeriph_Driver\src文件夹中，
Libraries\STM32F10x_StdPeriph_Driver\inc文件夹里面是它们的头文件。

9. stm32f10x_it.h、stm32f10x_it.c和stm32f10x_conf.h这三个文件是用户在编程时需要修改的文件，其他库文件一般不需要修改。如下图4-74所示

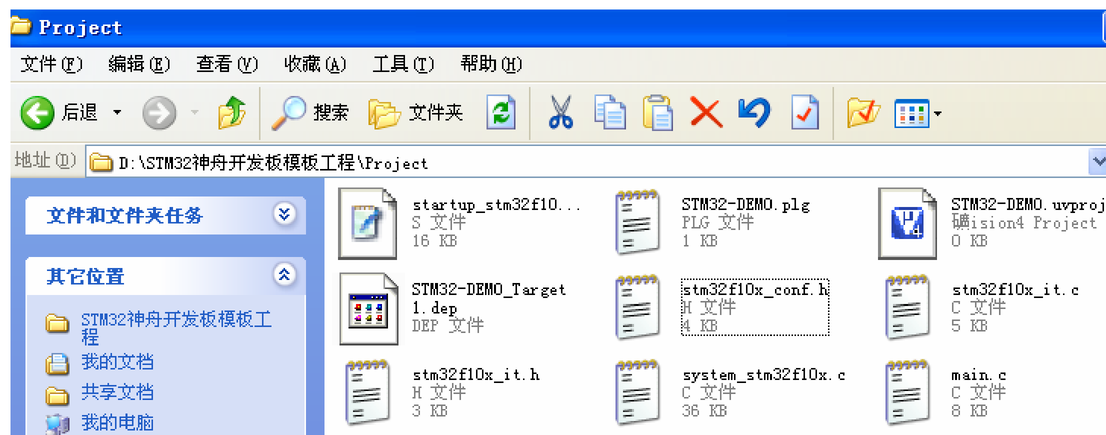


图 4-74 从库文件中拷贝需要的文件到文件夹指定位置

10. Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x\startup\arm文件夹下是用汇编写的启动文件。STM32F103C8T6开发板用的CPU是STM32F103C8T6，有128K Flash，属于中容量的，所以等下我们把startup_stm32f10x_md.s添加到我们的工程中。根据ST的官方资料：Flash在16~32 Kbytes为小容量，64~128 Kbytes 为中容量，256~512 Kbytes为大容量，不同大小的Flash对应的启动文件不一样，这点要注意。如图4-75所示

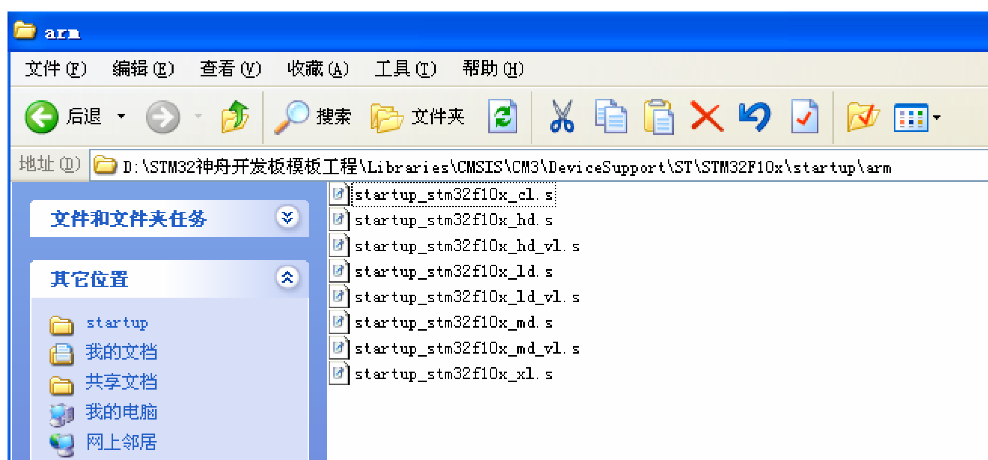


图 4-75 根据容量大小选择合适的启动文件

11.现在我们新建的工程目录如下图4-76所示：

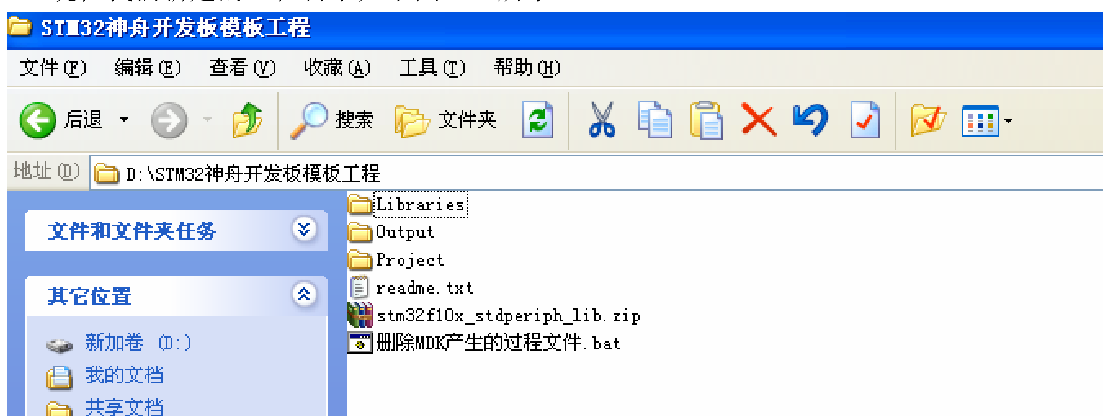


图 4-76 新建工程目录完成对库文件的移植

12.回到我们刚刚新建的MDK工程中，如下图4-77

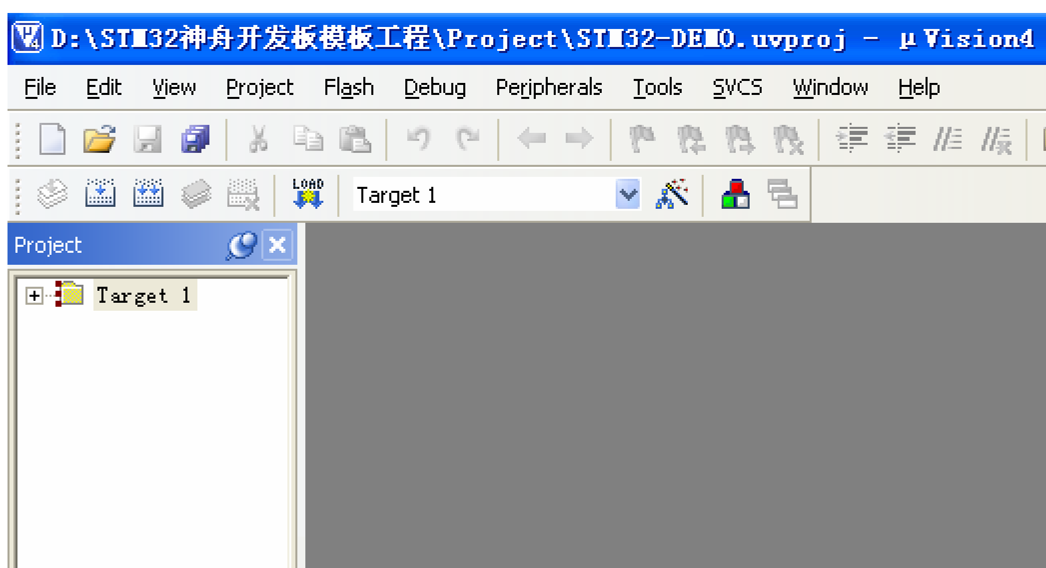


图 4-77 KEIL 新建立的工程页面

13.在STM32-DEMO上右键选中Add Group...选项，新建四个组，分别命名为start_code、project、libraries、CMSIS。start_code从名字就可以看得出我们是用它来放我们的启动代码的，project用来存放用户自定义的应用程序，libraries用来存放库文件，CMSIS用来

存放M3系列单片机通用的文件，如下图4-78所示：

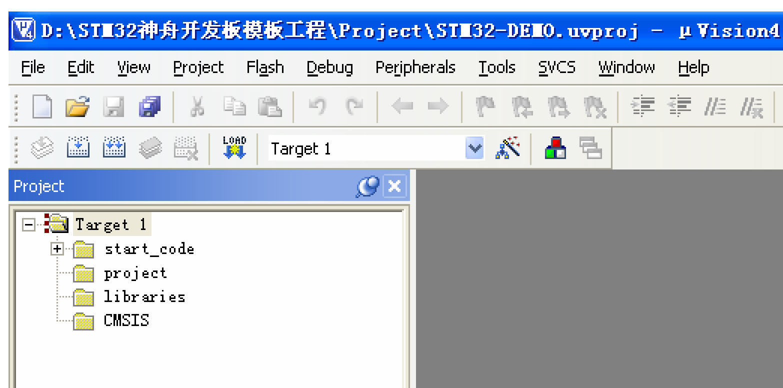


图 4-78 新建工作组

14.接下来我们往我们这些新建的组中添加文件，双击哪个组就可以往哪个组里面添加文件。我们在STARTCOKE里面添加startup_stm32f10x_md.s，在project组里面添加main.c文件和stm32f10x_it.c这2个文件，在libraries组里面添加“Libraries\STM32F10x_StdPeriph_Driver\src”里面的全部驱动文件，当然，src里面的驱动文件也可以需要哪个就添加哪个，这里将它们全部添加进去是为了后续开发的方便，况且我们可以通过配置stm32f10x_conf.h 这个头文件来选择性添加，只有在stm32f10x_conf.h文件中配置的文件才会被编译，。

15.我们首先在“Libraries/CMSIS/CM3/DeviceSupport/ST/STM32F10x”路径下添加system_stm32f10x.c文件(system_stm32f10x.c是ARM公司提供的符合CMSIS标准的库文件)；然后从Libraries\CMSIS\CM3\CoreSupport里添加core_cm3.c；注意，这些组里面添加的都是汇编文件跟C文件，头文件是不需要添加的。最终效果如下图4-79：

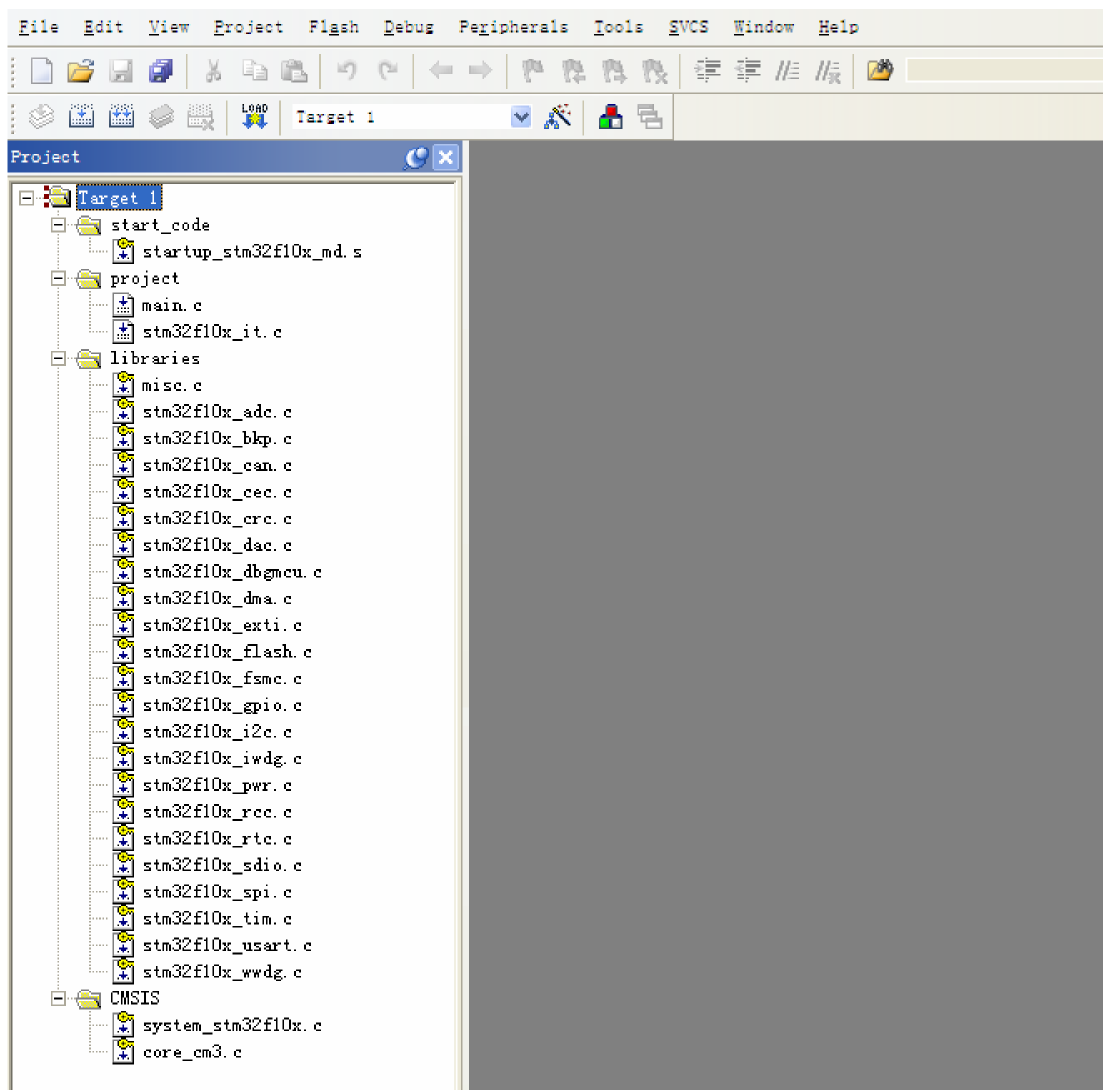


图 4-79 完成添加文件到我们的工程中

至于有些文件有个锁的图标，是因为这些都是库文件，不需要我们修改，属性为只读。

4.5.4 MDK环境设置

1. 经过以上的一些步骤，我们的工程已经基本建好，下面来配置一下MDK的配置选项，点击工具栏中的魔术棒按钮，在弹出来的窗口中选中“魔法棒”如下图4-80所示：

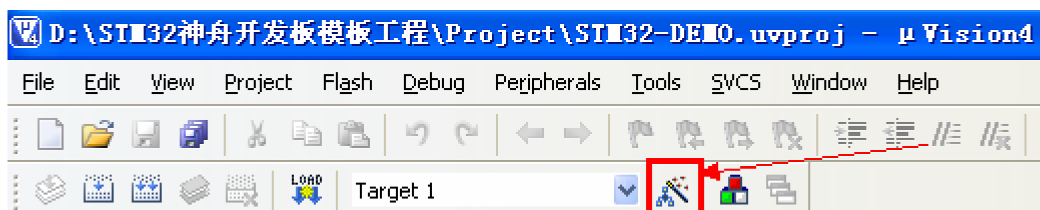


图 4-80 点击“魔术棒”进入配置设置页面

进入配置选项后选择“Output”如图4-81所示：



图 4-81 进入“Output”选项卡

步骤1：点击Select Folder for Objects... 设置编译后输出文件保存的位置，我们选择Output文件夹。如图4-82所示：

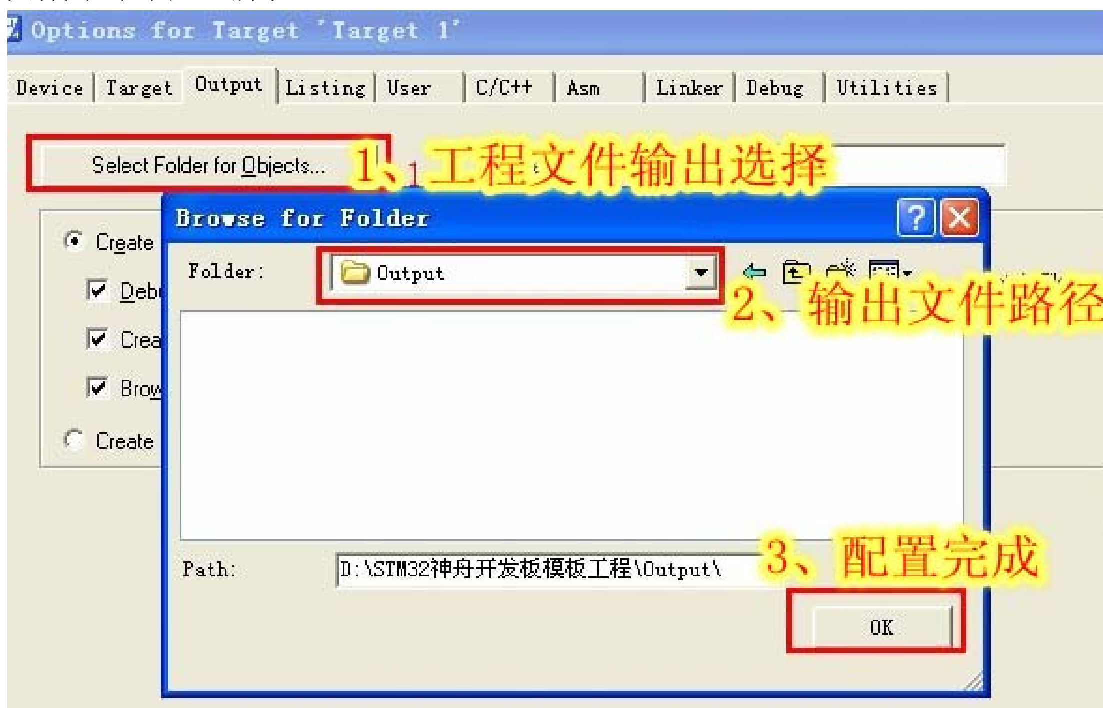


图 4-82 选择工程输出保存路径

步骤2：把编译好的输出文件名定为STM32-DEMO，如图4-88所示

步骤3：把 Create HEX File这个选项框也选上，表示编译输出HEX文件如图4-88所示

2. 选中选项卡“C/C++”，在Define 里面输入添加“USE_STDPERIPH_DRIVER, STM32F10X_MD”，如图4-83所示。

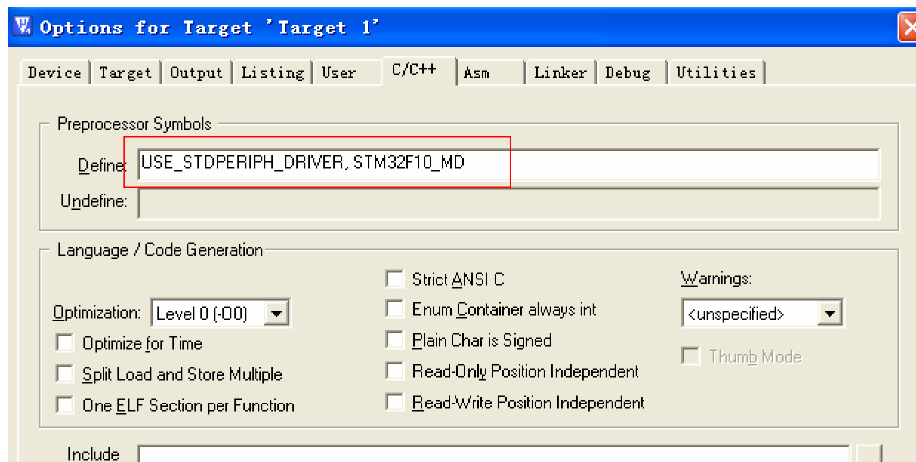


图 4-83 添加屏蔽编译器的默认搜索路径

添加USE_STDPERIPH_DRIVER是为了屏蔽编译器的默认搜索路径，转而使用我们添加到工程中的ST的库，添加STM32F10X_MD是因为我们用的芯片是中容量的，添加了STM32F10X_MD这个宏之后，库文件里面为大容量定义的寄存器我们就可以用了。芯片是小或大容量的时候宏要换成STM32F10X_LD或者STM32F10X_HD。其实不管是什么容量的，我们只要添加上STM32F10X_HD这个宏即可，当你用小或者中容量的芯片时，那些为大容量定义的寄存器我不去访问就是了，反正也访问不了。

3. 在“Include Paths”栏点击 ，在这里添加库文件的搜索路径，这样就可以屏蔽掉默认的搜索路径，如图4-84所示：

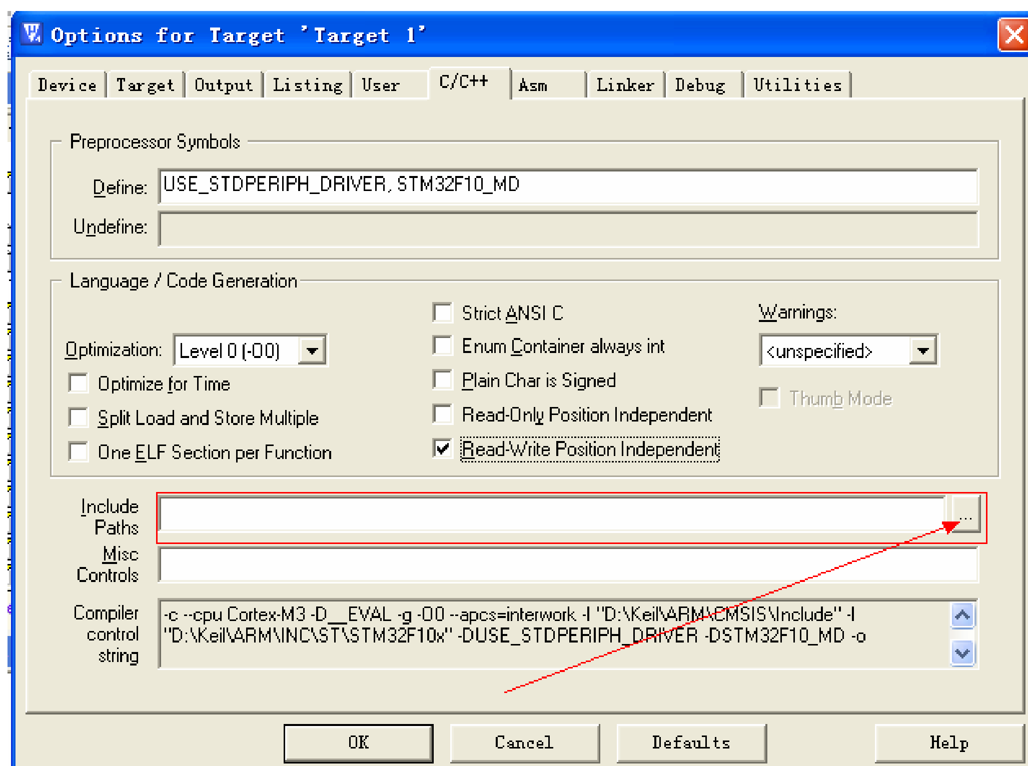


图 4-84 库文件的搜索路径

开始添加我们的搜索路径，路径如图 4-85 所示：

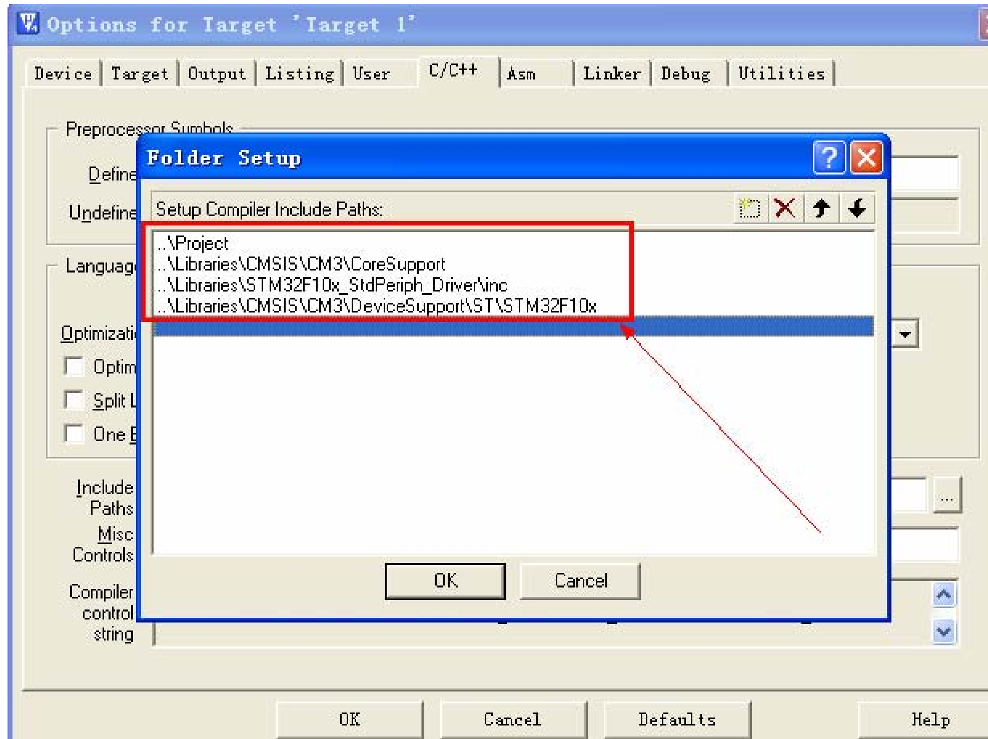


图 4-85 添加搜索路径

但当编译器在我们指定的路径下 搜索不到的话还是会回到标准目录去搜索，就像有些 ANSIC C的库文件，如 stdin.h 、stdio.h。

库文件路径修改成功之后如下图4-86所示：

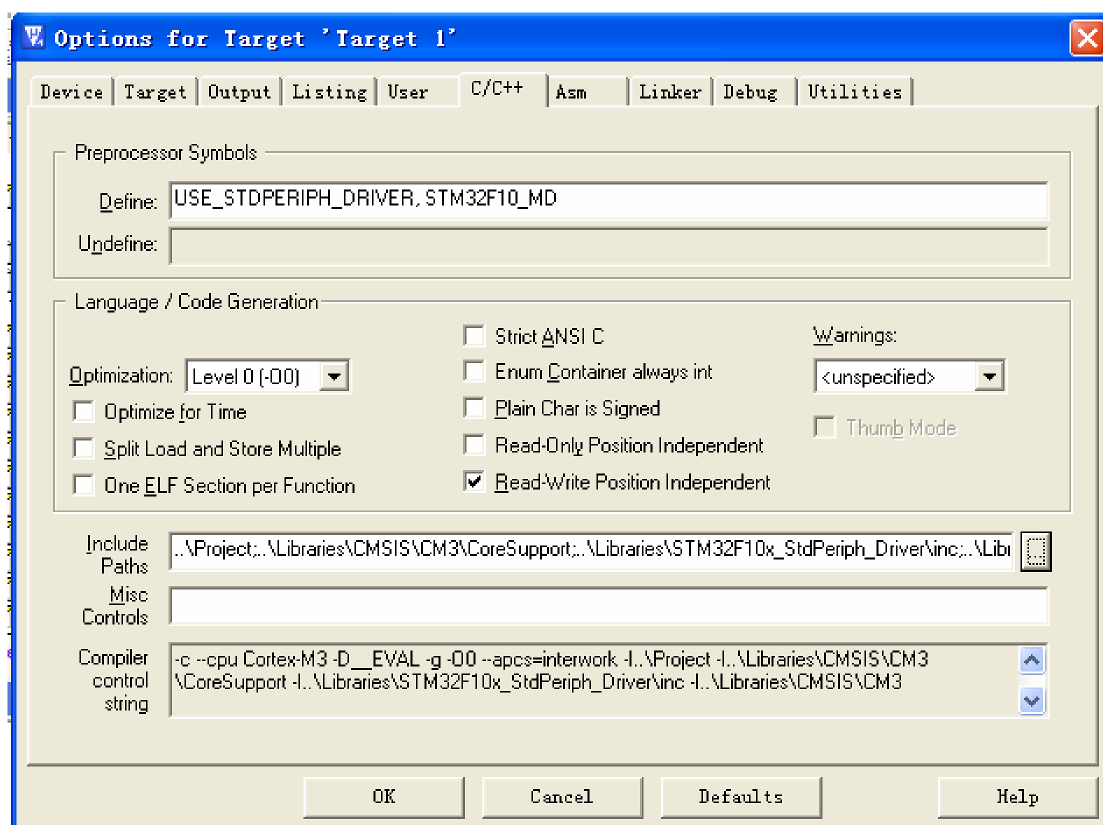


图 4-86 完成路径的修改

至此，我们的工程模板就建成了。学会新建工程，是学习stm32的第一步。 在main.c里输入下面的代码，并保存，然后编译代码，如图4-87所示：

```
/*****************************************************
* 内容：STM32神舟开发板模板工程
* 作者：www.armjishu.com
* 版本：v1.0
*****************************************************/
int main(void)
{

}
```

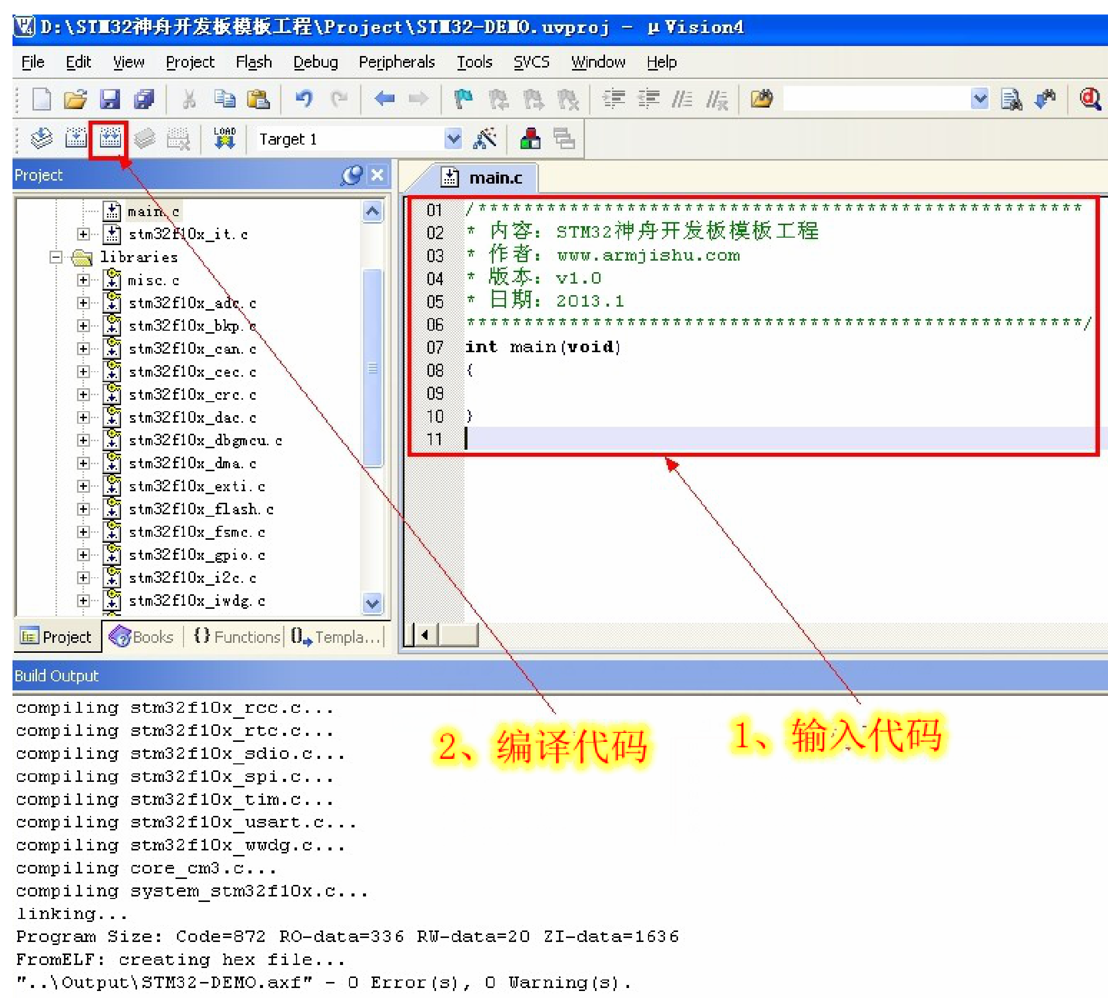


图 4-87 添加我们的代码并进行程序的编译

可以看到编译成功，最后产生了HEX文件，我们的模板搭建成功！

4.5.5 使用JLINK V8 仿真器硬件调试配置

1. 这个工程默认的是软件仿真，如果开发板要用J-LINK
调试的话，还需要在开发环境中做如下修改。实际上，我们开发程序的时候80%都是在硬件上调试的。

2. 点击，进入我们的配置选项，在 Debug 选项里进行一个配具操作，配置如下图 4-88 所示：

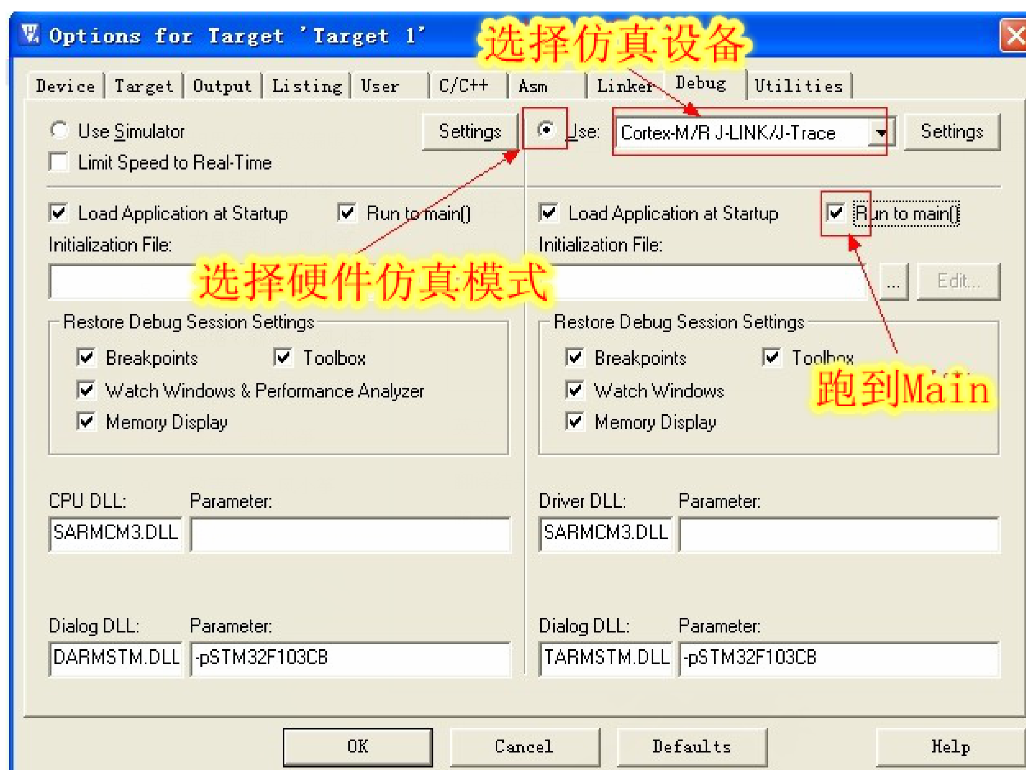


图 4-88 Debug 选项配置

3.点 Settings 按钮，查看是否识别到了目标板 CPU（注意，此处目标板应该上电，并将 JLINK V8 与模板连接好，JLINK V8 也需要上电），正常识别如下图 4-89 所示，设置完点 OK 退出。

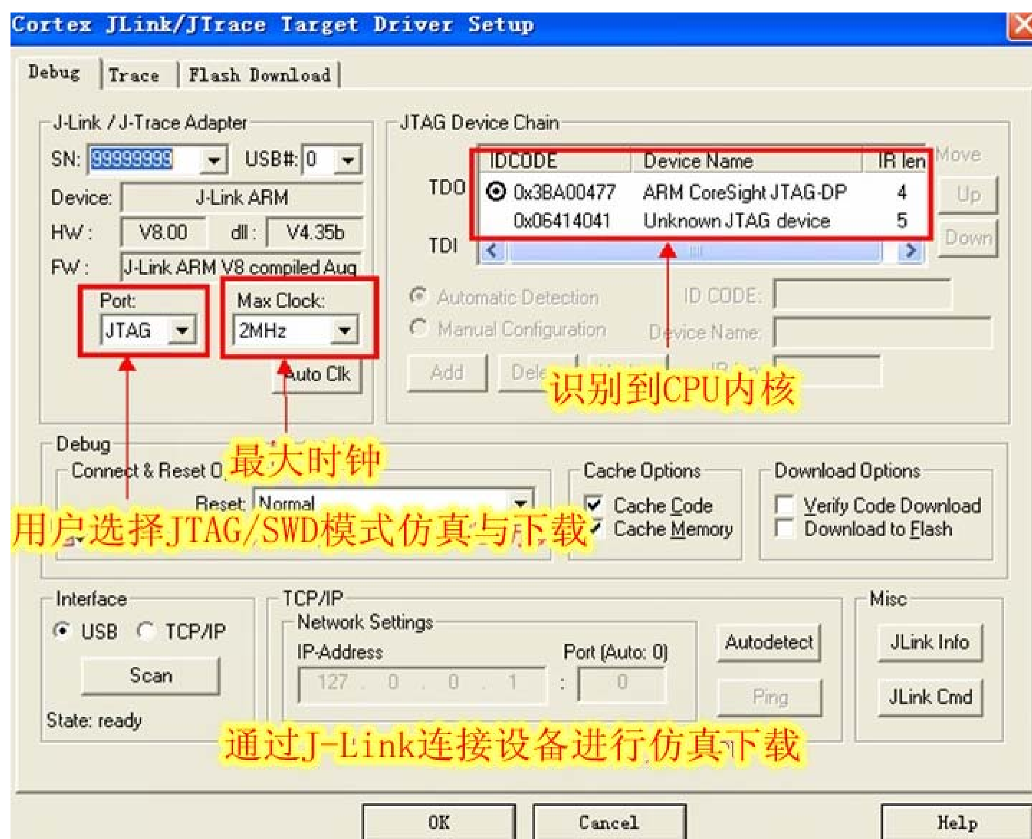


图 4-89 板子连接情况与模式的设置

4. 下面再选择Utilities选项卡，进行一个如图4-90的配置选择

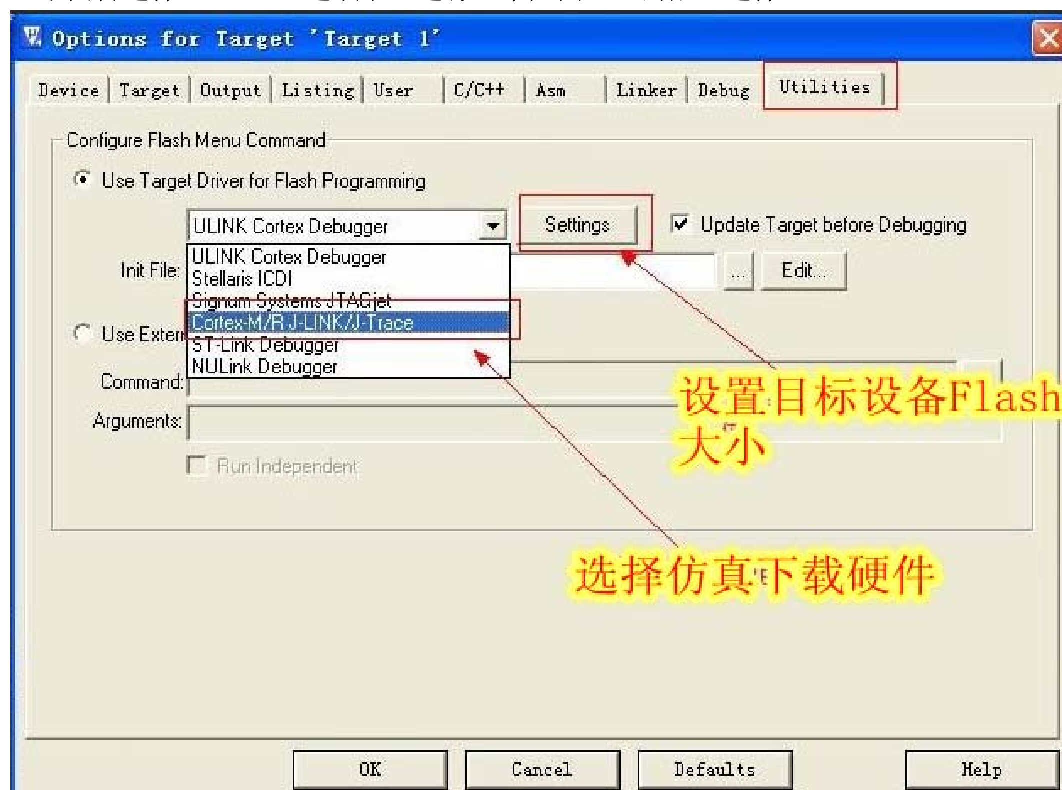


图 4-90 配置 Utilities 选项卡

5. 在选项卡Utilities\Setting\Flash download中我们设置成如下图4-91、4-92选择我们ARM的Flash大小：

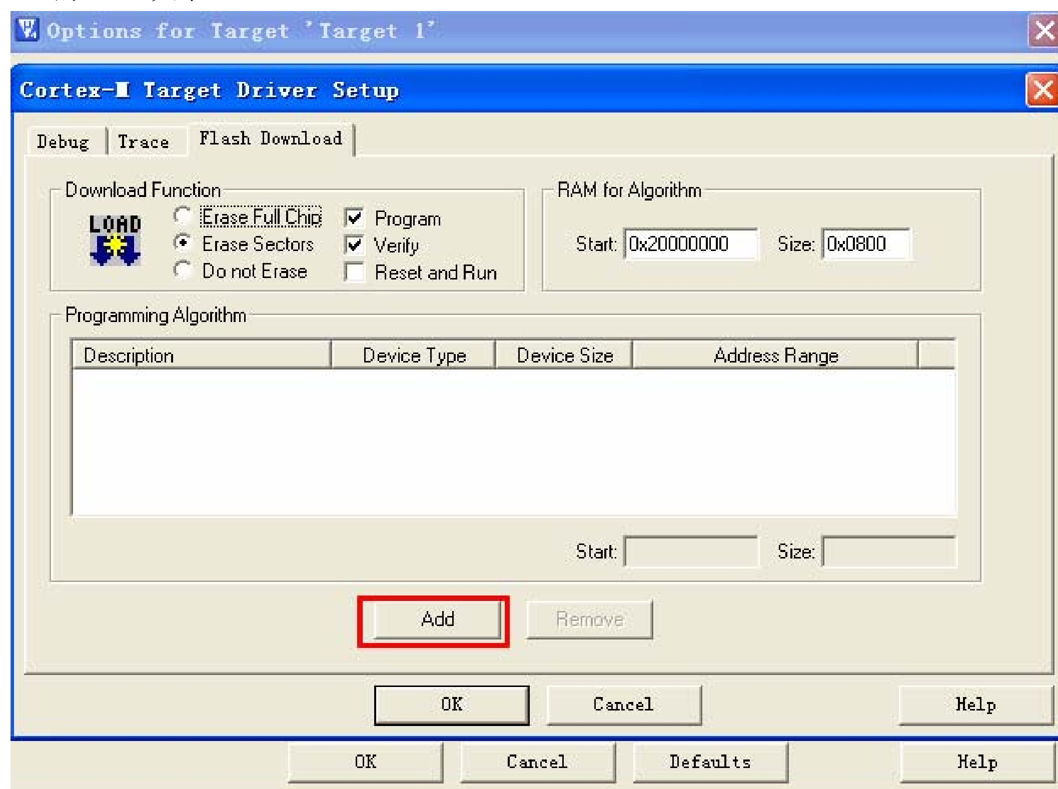


图 4-91 配置 Flash 大小

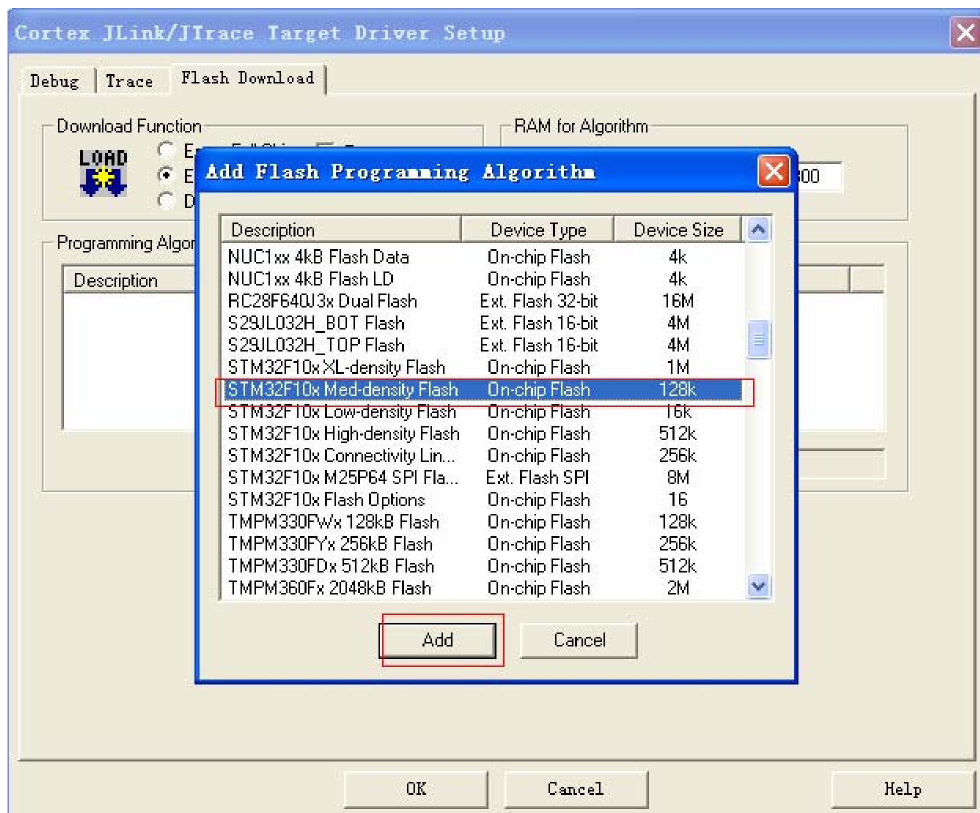


图 4-92 选择合适的 Flash 大小

配置完 Falsh 后点击“OK”退出配置，如下图 4-93 所示：

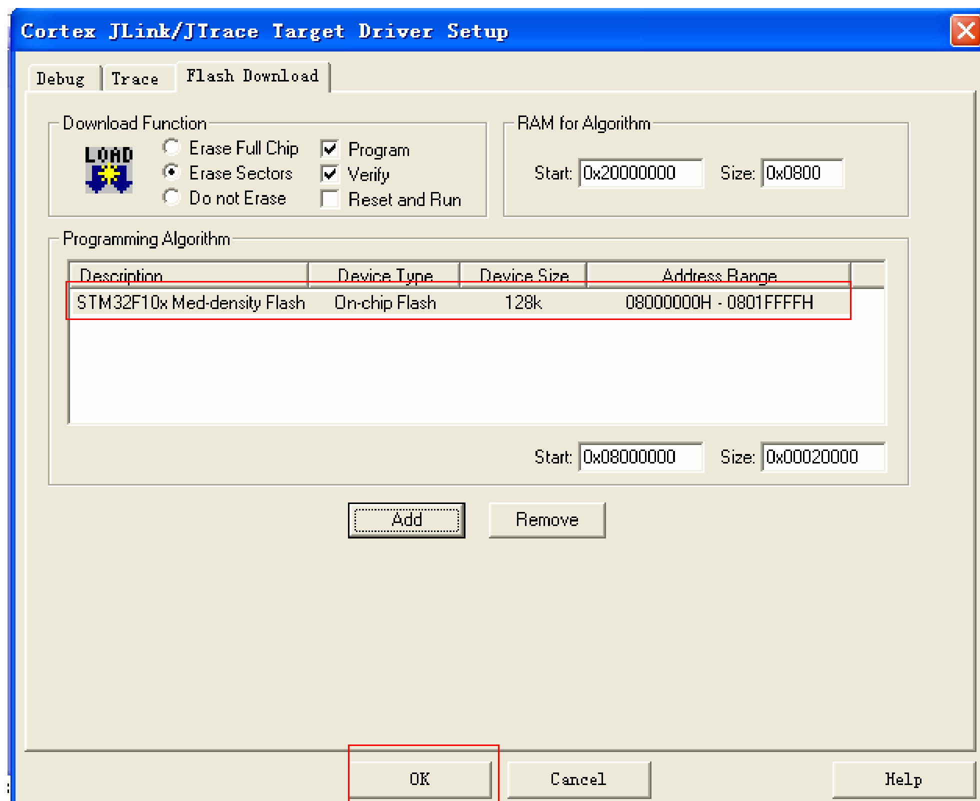


图 4-93 完成对 Flash 的配置

因为STM32F103C8T6芯片的内部ROM大小是64K点击OK按钮

6.编译全部代码，然后点LOAD下载程序到目标板如图4-94所示：

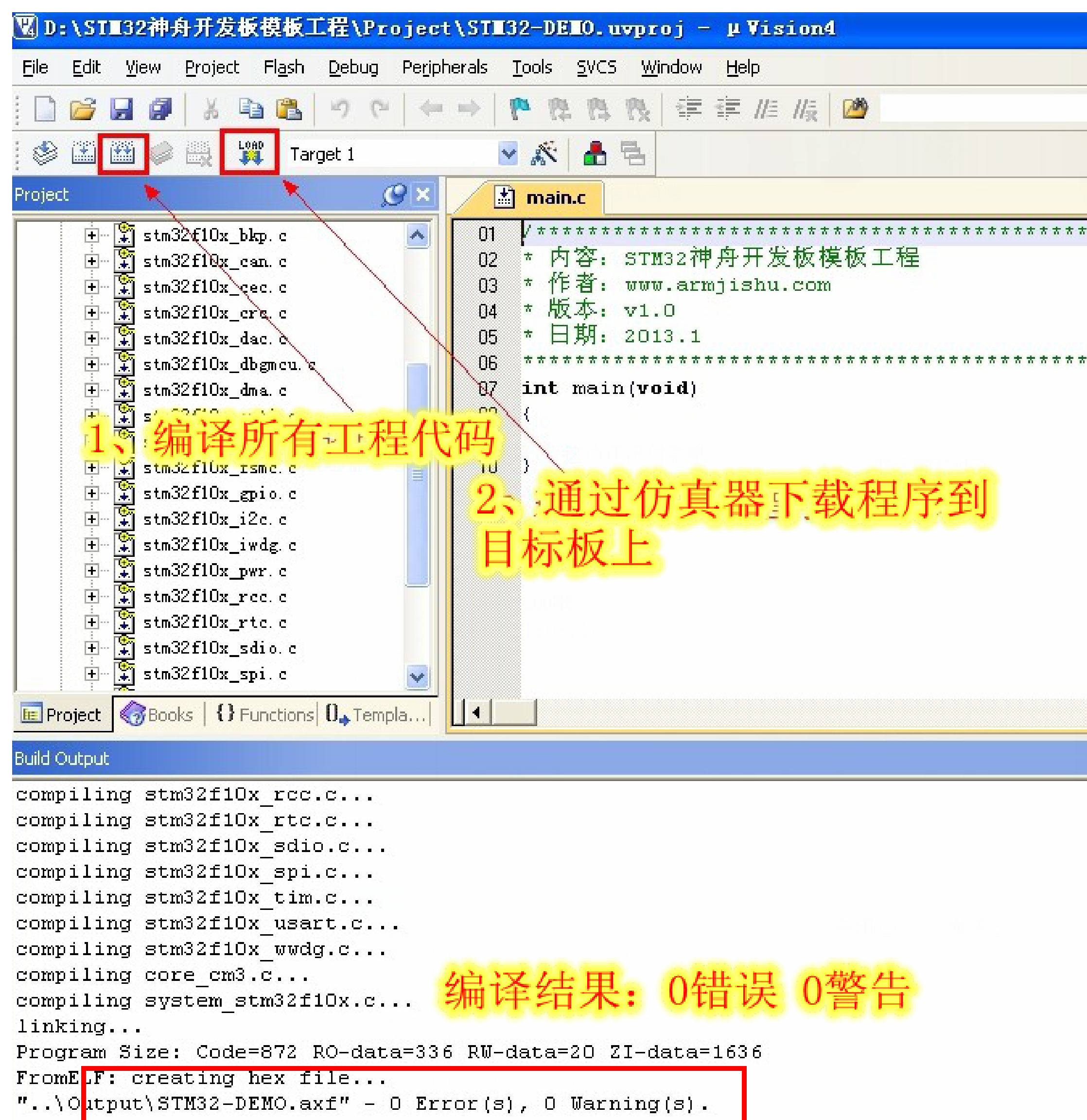


图 4-94 编译并 下载我们的工程

7. 到了这里就算是大功告成了。如果在新建工程中遇到什么问题，先不要急，可先参考STM32神舟系列的其他相关资料。

4.6 通过程序的分析总结 51 和ARM区别

通过上面的例程全面分析，可以感觉到其实 51 和 ARM 到最底层都是差不多的，这里主要是指，都是需要访问寄存器，改变寄存器的值来达到控制芯片工作的目的。

不同之处主要有：

1) ARM 比 51 的寄存器总数更多，毕竟 ARM 芯片的资源更多，所以就需要更多的寄存器进行管理。

2) ARM 比 51 单个功能模块的功能也越多，这一点拿 GPIO 举例，STM32 的 GPIO 管脚

有 8 种模式可以设置，而 51 却比较单一；主要是因为，ARM 具有丰富的接口，而这些接口有它独有的特性，而 ARM 又支持管脚灵活配置，可以用户自己在代码里指定，比如 AD/DA，比如中断等，而 51 一般都是指定这个管脚就是中断管脚，所以 ARM 因为增加了这些接口有它的特性，所以不得不增加一些 GPIO 管脚的模式类别让用户自己去配置，这也验证了一句话，功能越多，感觉越方便，实际上内部实现可能越复杂。

3) ARM 的性能要强过 51；假如说 51 只能打酱油的话，那么 ARM 既可以打酱油，又可以当搬运工搬砖头，力气比较大；这样 ARM 丰富的资源，就有更大的空间去控制和管理它，因此它的软件代码方面可能要比 51 复杂得多，ARM 速度快，可以做的事情非常多，那么很多的应用都可以往上面放；这里举个简单易懂的例子，假如 1 个用户用一个普通电脑就够用了，那么 2 个用户同时登录到一个强劲的电脑就够用了，如果出现一个超强的计算机电脑呢？它的速度足可以满足同时 100 个人使用，那么基于它之上的控制和协调这个 100 个人的工作不能相互冲突的功能也就要非常强大，不仅仅要硬件强大，硬件是基础，还要靠软件去把这个强劲的蛋糕合理的分配给每个人，那么这个软件必然就会变得比较复杂了。

第五篇 ARM 实战篇

5.1 神舟 51+ARM模块如何使用

5.1.1 神舟 51+ARM模块与最小系统有什么区别

在简单了解了什么是单片机之后，然后我们来构建单片机的最小系统，单片机的最小系统就是让单片机能正常工作并发挥其功能时所必须的组成部分，也可理解为是用最少的元件组成的单片机可以工作的系统。对 STM32 系列单片机来说，最小系统一般应该包括：单片机、时钟电路、复位电路、输入/输出设备等。

STM32F103 系列 ARM 的最小系统如下 5-1 图所示：

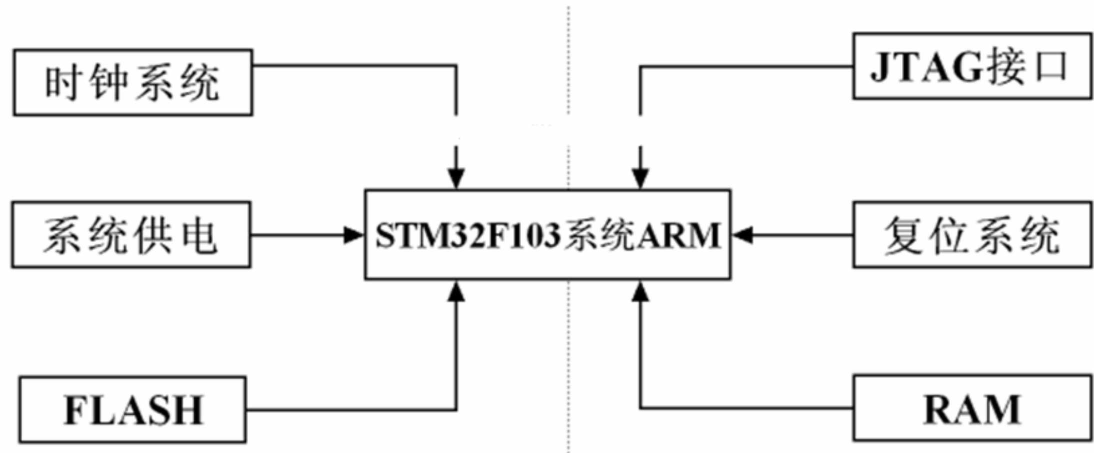


图 5-1 ARM 的最小系统框图

构建一个最小系统必须要有的是时钟，CPU 没有时钟不知道怎么一步一步的运行，要有复位系统，要有供电，没电更是跑不起来，这 3 条是必备的；至于下载接口，RAM 和 ROM（内部的 FLASH）这些也是 CPU 本身就具备的，芯片内部都现成有的。

下面是神舟 51+ARM 模块的图片如图 5-2 所示：

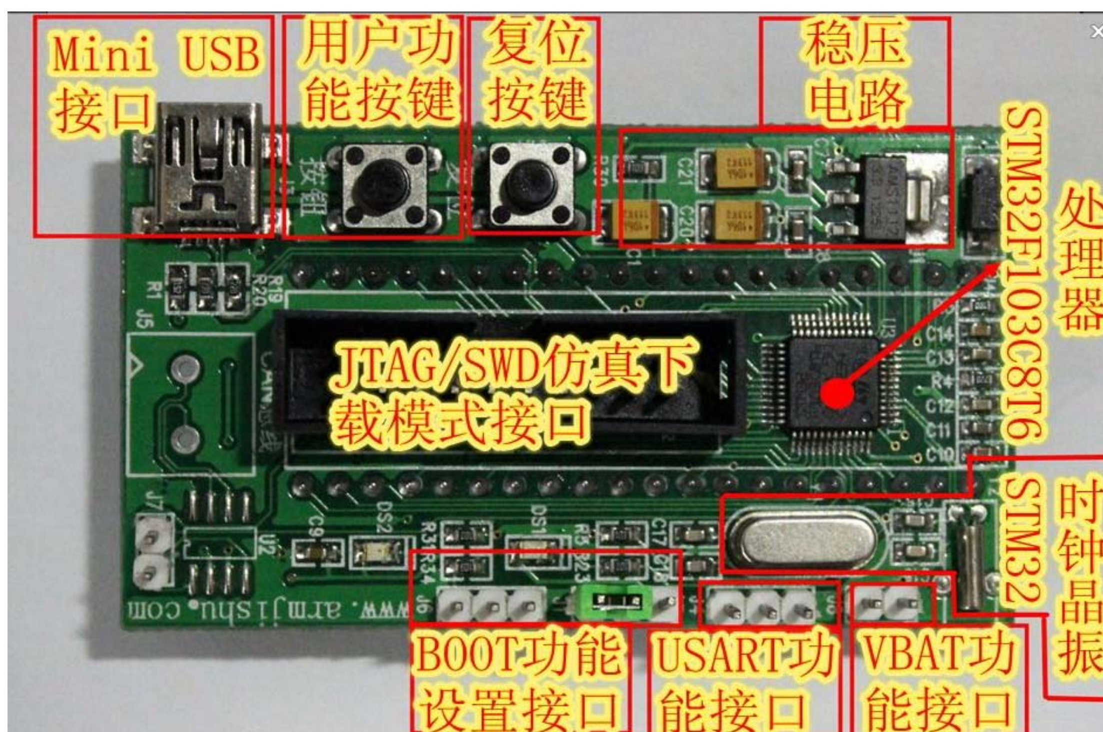


图 5-2 神舟 51+ARM 模块

神舟 51+ARM 小模块的硬件电路对比最小系统来说，多了许多外围电路，增加了如下：

增加 1：比如 STM32F103C8T 内部本身已经具有了复位电路，但是在外部电路中，也增加了一套复位电路，并且有独立的复位按键。

增加 2：STM32F103C8T 虽然有内部时钟，但是在外部增加了一个 8M 的时钟晶振提供稳定的时钟，而且当外部时钟出现故障的时候，会自动切换到内部时钟电路。

增加 3：最小系统正常工作的时候，内部也有实时时钟，但 51+ARM 模块上也更新了一个 32.768k 的实时时钟晶振。

增加 4：增加了 STM32 的 BOOT 引导模式，方便设置不同的引导模式采取不同的下载方式。

增加 5：增加了一个串口功能接口，方便打印和调试。

增加 6：增加 VBAT 功能，可以为实时时钟接上电池，即使芯片掉电，实时时钟也是正常供电的。

增加 7：增加了一个按键，方便按下去

增加 8：增加了 CAN 通信接口，可以进行 CAN 通信

增加 9：增加了 USB 接口，可以通过 USB 接口对板子进行供电。

可以看到神舟 51+ARM 模块增加了这么多功能，与最小系统相比，更加完善了，把芯片的一些外围接口也进一步扩充出来了。

5.1.2 如何把ARM模块扣在神舟 51 单片机板子上

神舟 51+ARM 单片机为了系统的多样化，单片机硬件上还配置了一个 40Pin 的芯片卡座，根据所接芯片的不同，实现 51 到 ARM 或者是 ARM 到 51 之间的转变，如图 5-36 所示：

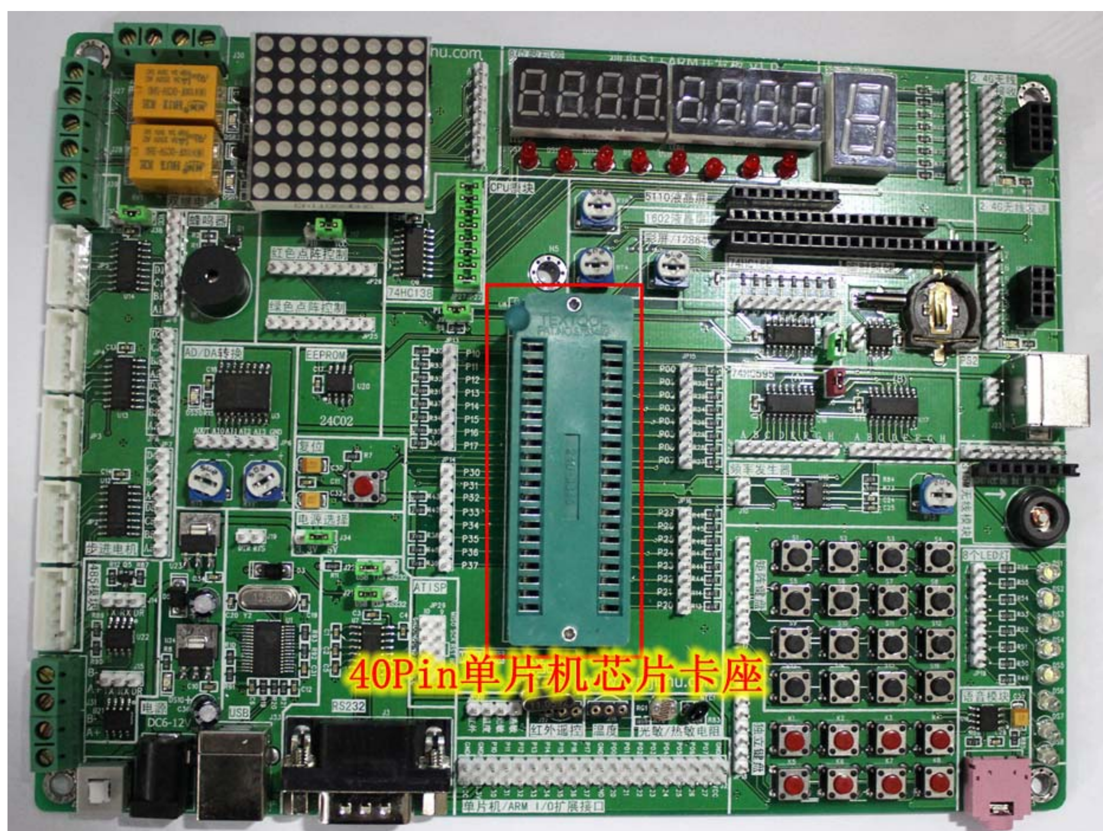


图 5-3　40Pin 的芯片卡座

当我们需要使用 51 系统的时候，只需要在这个芯片卡座上接上 51 单片机芯片，把 51 单片机芯片，如图 5-4 放在这个 40PIN 的绿色卡座上，将左上边的手柄按下去，使得座子将芯片夹紧（注意，51 单片机芯片有一个凹口，顶端凹点左处就是单片机芯片的第 1 脚），注意方向不要放反了。下图 5-5 为单片机芯片的外形图与板子接上 51 单片机芯片后的 51 板子。



图 5-4 STC51 单片机芯片

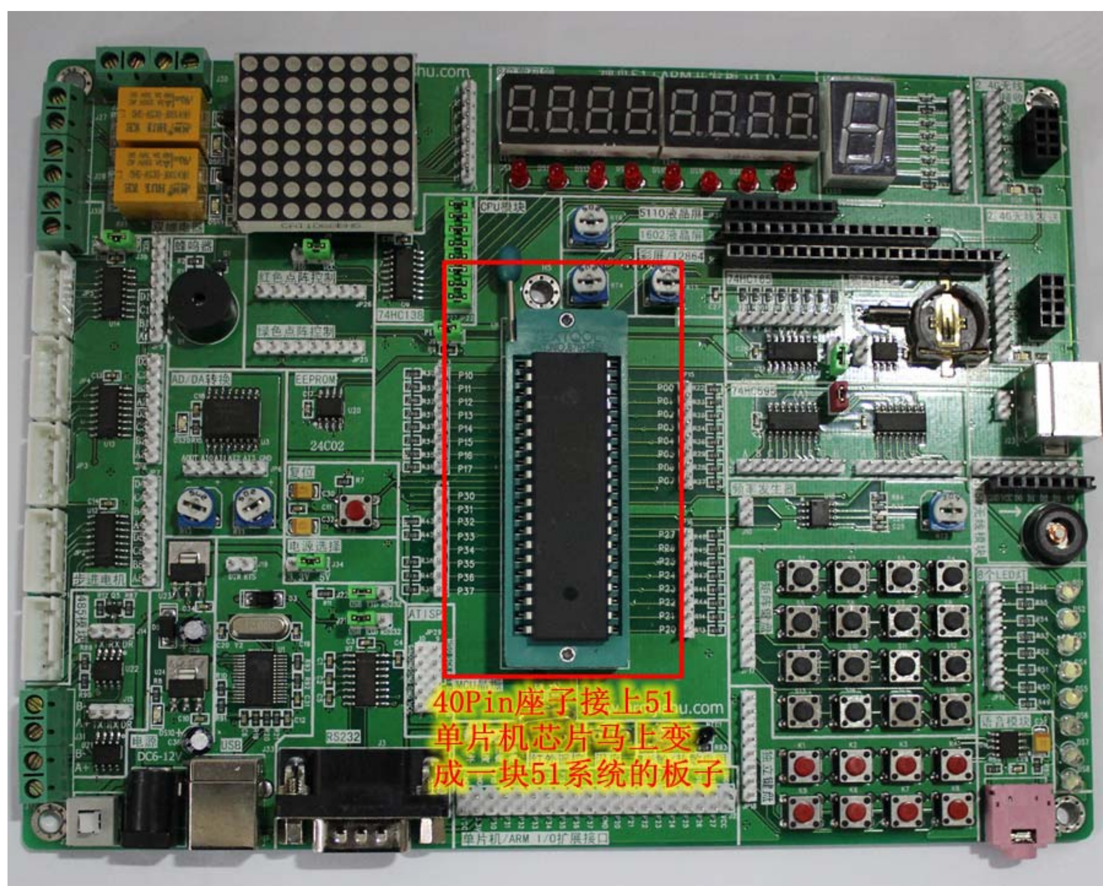


图 5-5 单片机板子安装 51 系统的芯片

即可让该板子成为一块 51 单片机板子，下载 51 程序到单片机上，通过信号线的连接，即可控制板子上的外围器件使其工作，需要注意的是芯片的正确接法。

既然是 51+ARM 单片机的板子，不可能只是实现 51 单片机系统的吧。ARM 系统如何而来呢？别急，下面我们马上让这个 51 单片机的板子变成 ARM 系统的单片机板子，方法很简单，只需要在 40Pin 的芯片卡座上把 51 单片机的芯片替换成我们的 ARM 系统的核心板子就好了。因为芯片和座子都是 40 个管脚的，所以把座子上的 51 单片机拿开，如果放上一个 40 个管脚的 ARM 模块呢？如图 5-6 所示：

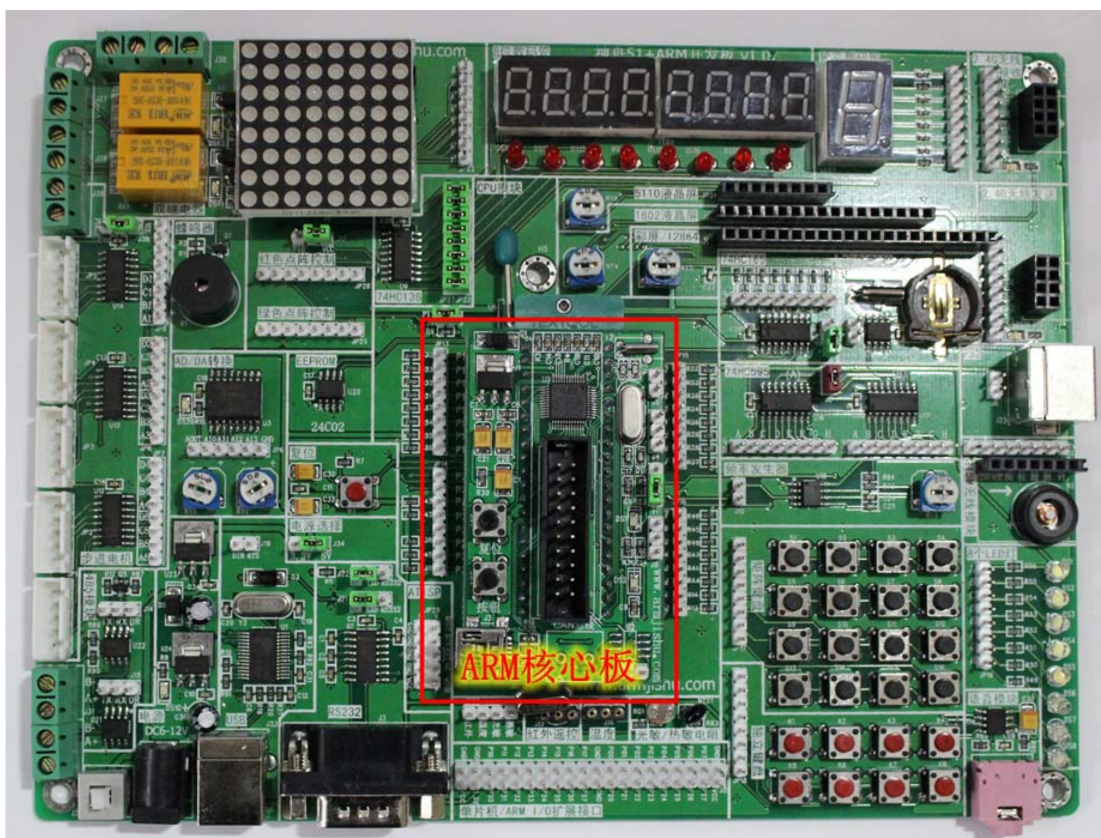


图 5-6 单片机板子安装 ARM 核心板

这样就变成了一款 ARM 的开发板了，很神奇吧，大家再看一张图片，如图 5-7，之前的我们设计的一个 ARM 芯片空板扣上去会是什么情况？



图 5-7 ARM 系统小板

把这个 ARM 系统小板安装到我们的单片机上后，如图 5-8 所示。对，又变成了一款 ARM 开发板了，这两个 ARM 板有什么区别呢？区别主要是这 2 个核心板引出的管脚不同，其次是这两个最小核心板上的电路不相同。

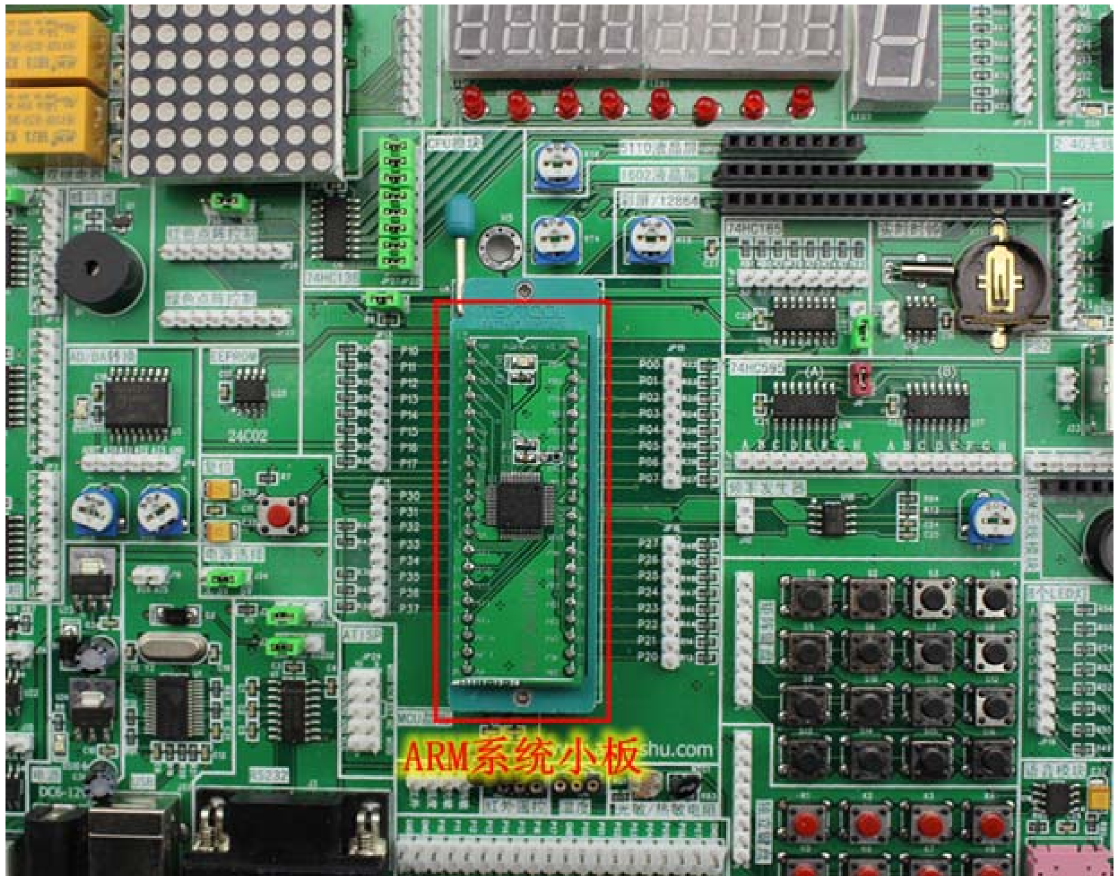


图 5-8 单片机安装 ARM 系统小板

所以 ARM 核心板目前我们有 2 种，一种是只是单独把 ARM 单片机芯片的引脚引出来的，另外一种是带各种功能的，下载程序的 JTAG、BOOT 的功能设置接口、电源电路与复位功能按键电路等等，我们接下来要从 51 过渡到 ARM 的这个过程，主要就是介绍这种比较完善的 ARM 核心板模块。

6.1.3 扣上ARM模块后 51 单片机板上的原理图怎么看

1. 解读 51 单片机处理器的原理图

原理图，顾名思义就是表示电路板上各器件之间连接原理的图表。在方案开发等正向研究中，原理图的作用是非常重要的，而对原理图的把关也关乎整个项目的质量甚至生命。由原理图延伸下去会涉及到 PCB layout，也就是 PCB 布线，当然这种布线是基于原理图来做成的，通过对原理图的分析以及电路板其他条件的限制，设计者得以确定器件的位置以及电路板的层数等。

下面开始分析一下 51 单片机的原理图，实物图如图 5-9 所示，电路原理图如图 5-11 所示：

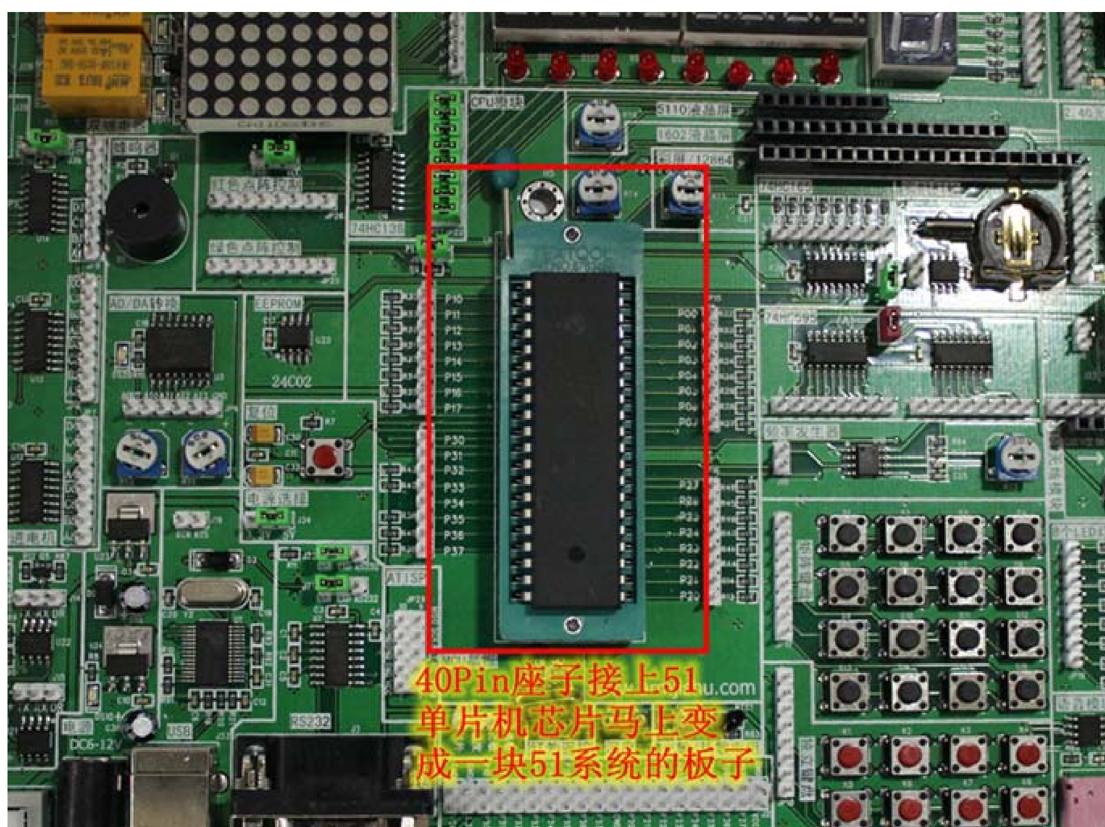


图 5-9　51 单片机实物图

原理图如图 5-10 所示：

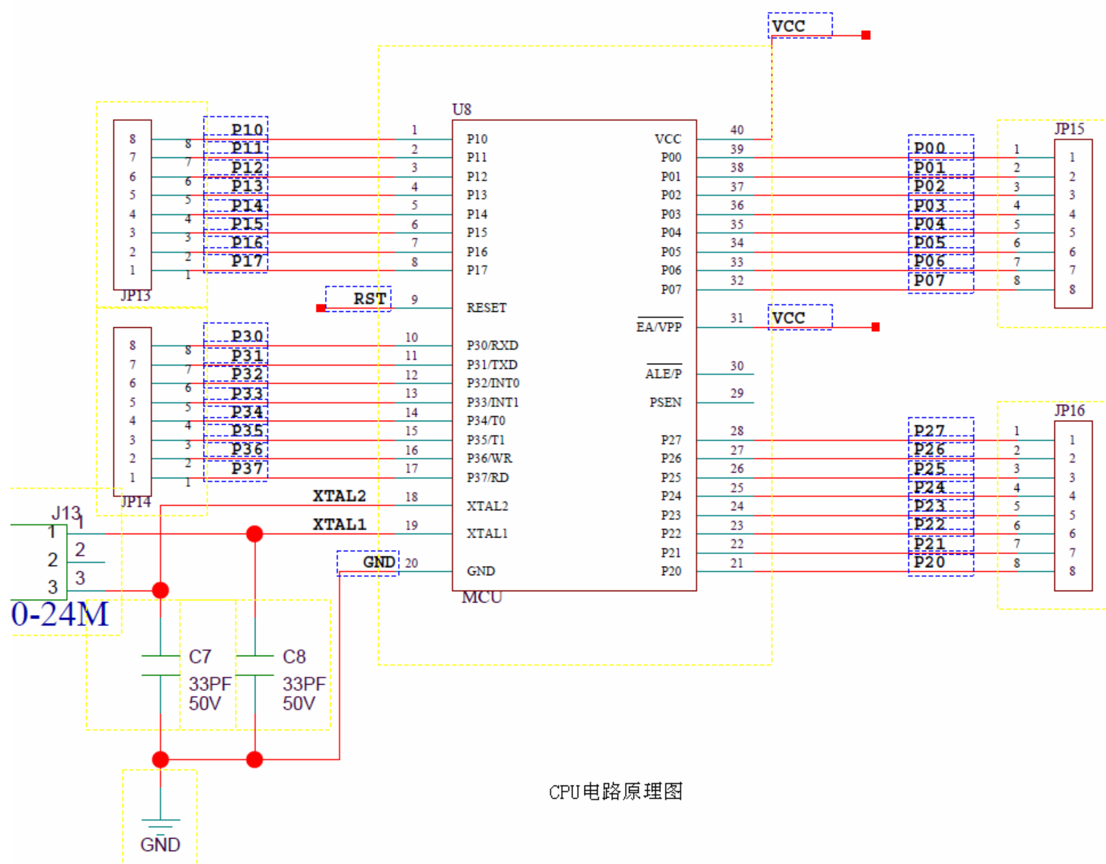


图 5-10 51 单片机处理器部分原理图

从图 5-10 的原理图中可以看出，原理图芯片左边从 1-20 有 20 个管脚，第 1 脚就是 51 单片机的 P10，第 1 脚到第 8 脚是 P1 端口的 8 个管脚；第 9 脚是 RST 复位管脚，第 10 脚到第 17 脚是 P3 端口的 8 个管脚，第 18、19 脚是 XTAL1 和 XTAL2 晶振管脚，第 20 脚是接 GND 的管脚。

原理图芯片右边从 21-40 也有 20 个管脚，其中第 21-28 是 P2 端口的 8 个管脚，其中 29 脚和 30 脚悬空，第 31 脚接 VCC 管脚，其中第 32-39 是 P0 端口的 8 个管脚，第 40 脚接 VCC 管脚。

处理器有 P0、P1、P2 和 P3 共 4 路的 I/O 口，分别接到 JP15、JP13、JP16 和 JP14 共 4 个座子上，可以通过排线的连接座子供到需要控制的外围电路上去控制它。在原理图中还有一个晶振电路，18、19 脚通过一个 JP13 的座子接一个 0 到 24M 的晶振，方便更换。我们的板子使用的是一个 11.0592MHz 的晶振，图中还有复位引脚与电源。

2. 解读 ARM 模块扣到神舟单片机底板上的处理器的原理图

从上面的分析可以同样分析一下 ARM 的原理图，先看下我们的 ARM 系统单片机板子的实物图，如图 5-11 所示：

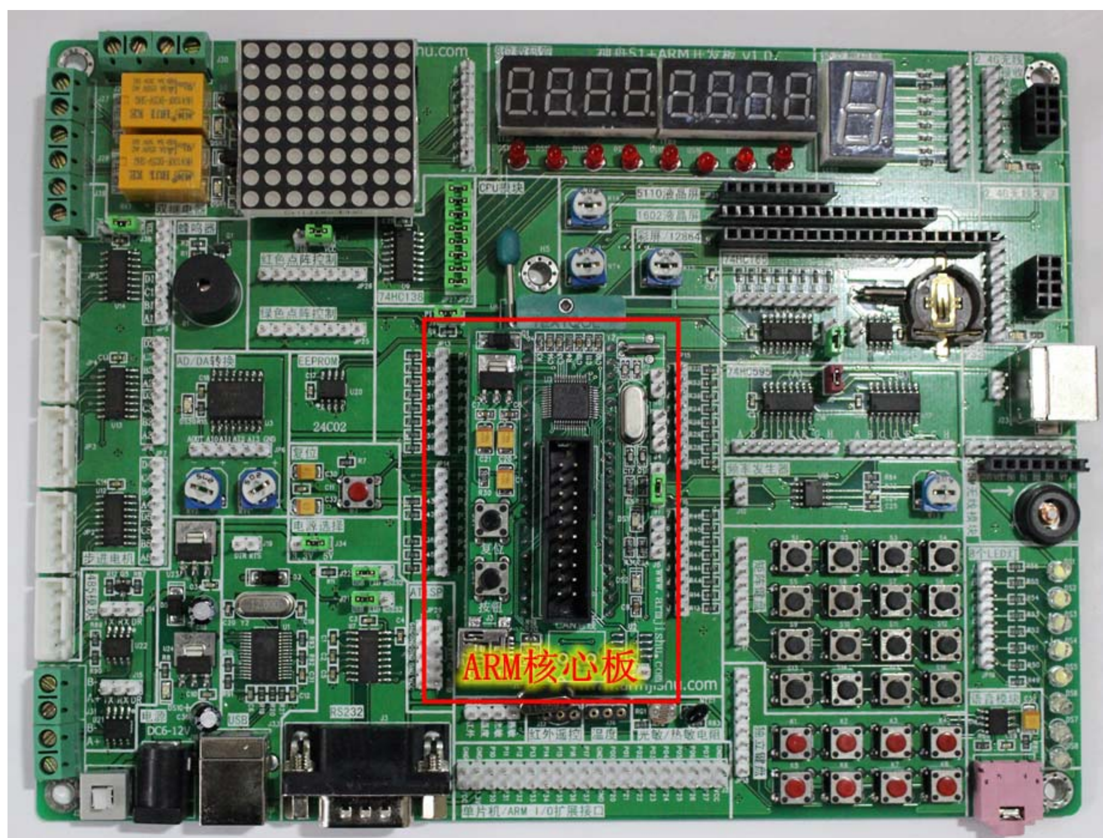


图 5-11 ARM 系统单片机板子实物图

电路原理图如图 5-12 所示：

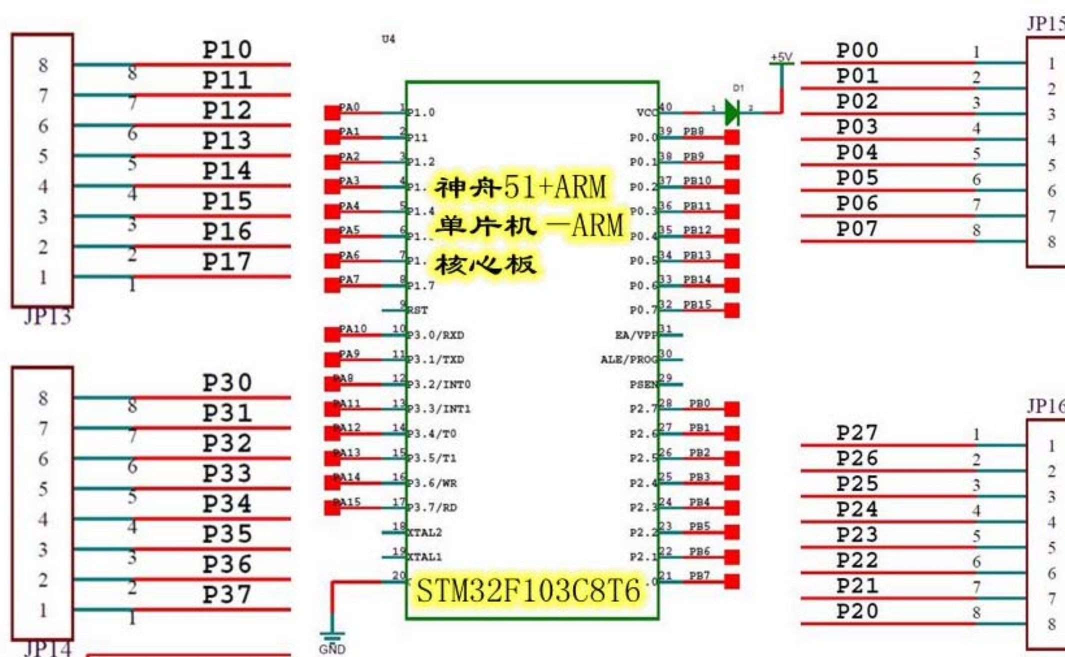


图 5-12 ARM 系统单片机板子处理器原理图部分

上图为神舟 51+ARM 之 ARM 模块的 CPU 的电路原理图，也就是 STM32F103C8T6 的芯片管脚引出 40 个管脚来；从原理图中可以看出，原理图引出的管脚左边从 1-20 有 20 个

管脚，第 1 脚就是 STM32F103C8T 芯片的 PA0，第 1 脚到第 8 脚是 PA 端口的前 8 个管脚 PA0-PA7；第 9 脚原本是 51 单片机接复位管脚，这里也悬空，防止被影响到，第 10 脚到第 17 脚是 PA 端口的后 8 个管脚 PA8-PA15，第 18、19 脚是 51 单片机的 XTAL1 和 XTAL2 晶振管脚，在这里接 ARM 板的时候悬空，否则可能会干扰到 ARM 板芯片的，第 20 脚是接 GND 的管脚。

原理图芯片右边从21-40也有20个管脚,其中第21-28是PB端口的前8个管脚PB0-PB7，其中 29 脚、30 脚，第 31 脚都悬空，没有任何的作用，其中第 32-39 是 PB 端口的后 8 个管脚 PB8-PB15，第 40 脚接 VCC 管脚、但是因为 51 单片机是 5V 供电的，而 ARM 是 3.3V 供电的，所以在 40 脚这个位置增加了一个稳压二极管，将电压稳定在 3.3V 给到 ARM 板。

所以总结来说，ARM 板的处理器有 PA0~PA15、PB0~PB15 总共 32 个管脚，分别接到 JP15、JP13、JP16 和 JP14 共 4 个座子上，可以通过排线的连接座子供到需要控制的外围电路上去控制它。

5.2 神舟 51+ARM模块的硬件电路分析原理图

5.2.1 神舟 51+ARM的原理图

神舟 51+ARM 核心板原理图见产品的光盘资料，打开我们光盘中的原理图文件：**\51 单片机资料\神舟 51+ARM 光盘资料\步骤 2 神舟 51 开发板电路图、安装图**路径下的 ARM 核心板原理图，如图 5-13 所示：

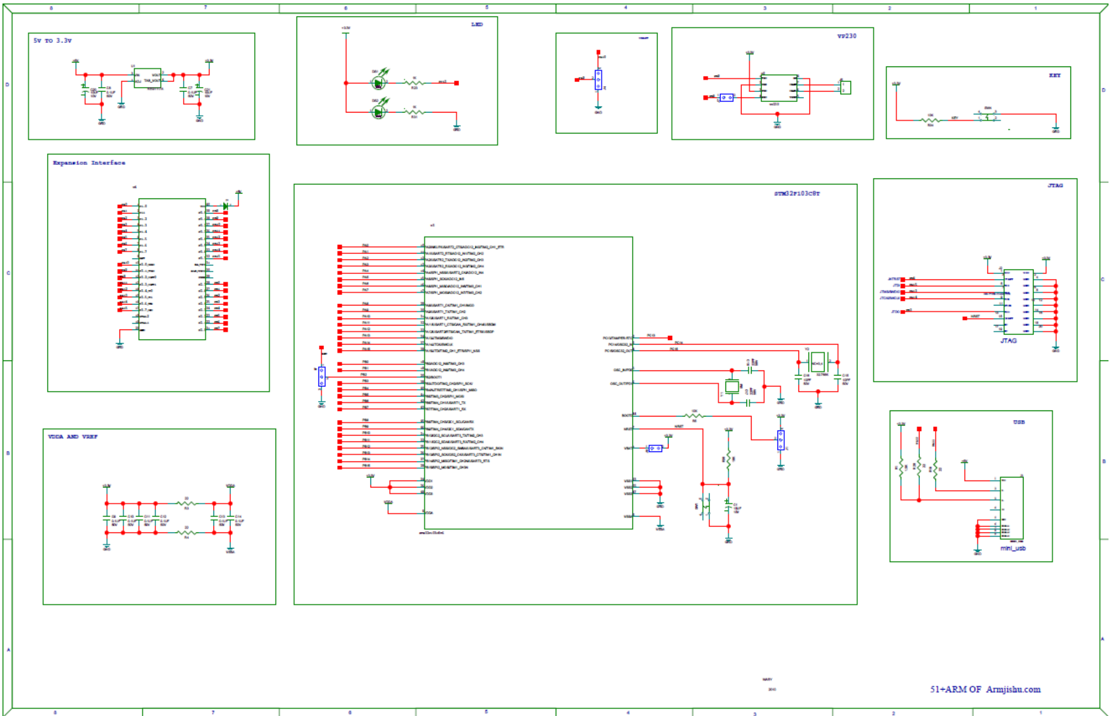


图 5-13 ARM 核心板原理图

5.2.2 神舟 51+ARM的功能特点

神舟 ARM 核心板是一款基于 STM32F103C8T6 的开发板，面向学生以及想从事 STM32 开发的工作者等广大爱好者，性价比非常高。

神舟 STM32-ARM 核心板的产品外观及对应各功能如下图 5-14 所示：

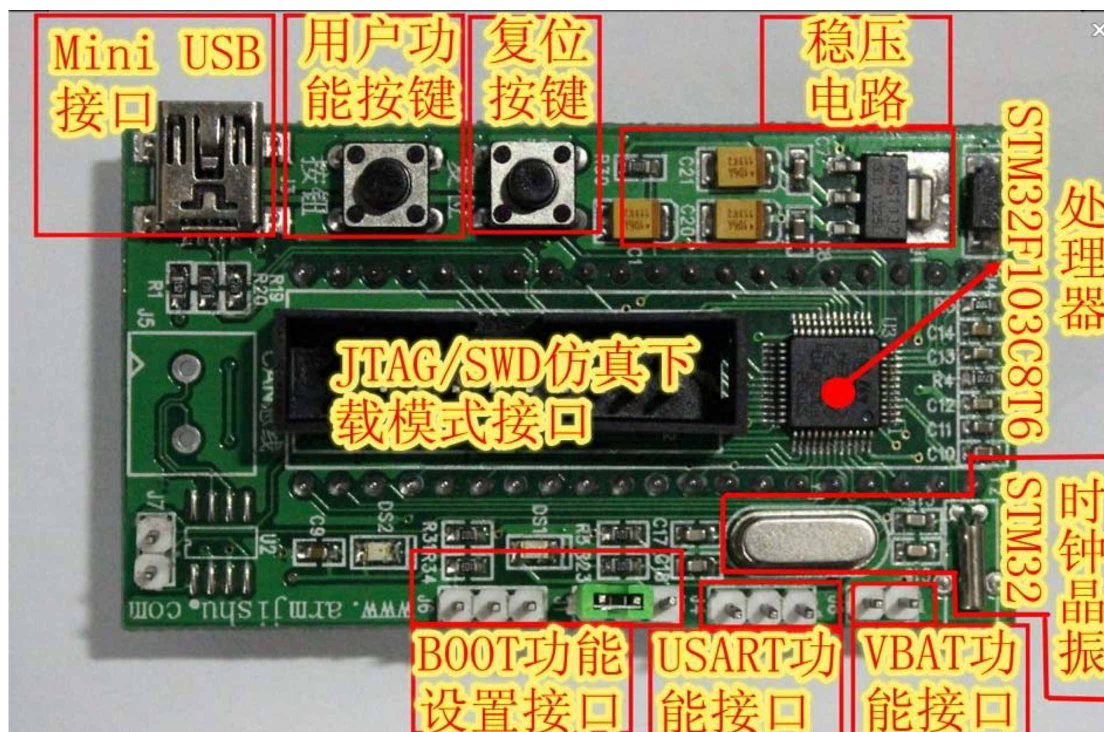


图 5-14　ARM 核心板产品外观及对应各功能

- ◆ 供电电源
- ◆ STM32F103C8T6，
- ◆ 1 个电源指示灯（绿色）
- ◆ 1 个用户状态指示灯（绿色）
- ◆ 1 个 USART 转换接线接口
- ◆ 1 个复位按钮，控制整板硬件复位
- ◆ 1 个用户功能按钮
- ◆ 1 个标准的 JTAG/SWD 仿真调试下载接口
- ◆ 1 个 USB 全速接口

从上面的板载资源可以看出，ARM 核心板板载资源虽然简单，但其实也可以制造出许多非常丰富的例程来，这些基本包括 STM32 爱好者常用的硬件资源，尤其是从 51 入门到 STM32 的爱好者能得到非常多的帮助。此外，核心板还将处理器所有 GPIO 接口通过双排插针接口引出，非常方便产品的功能扩展和其他功能模块调试，让你的开发变得更加简单。

核心板的特点有：

1）外观小巧。

2）性价比高。功能强大，但价格便宜

3）设计灵活，核心板上除了晶振外所有的 GPIO 口通过双排插针全部引出，可以通过外接一些模块，增加板子的功能

4) 资源丰富，小巧而精致

5）调试方便。与主流调试仿真工具 J-Link V8 完美结合，让您快速找到代码的 BUG。

6）提升空间。如果由 ARM 核心板学习完后，还可以考虑直接升级到神舟 I 号、神舟 III 号，神舟 IV 号，内部资源代码都是可以相互借鉴，为你提供更多丰富的资料，方便学习。

6.2.3 STM32F103C8T6 处理器

核心板使用了 STM32F103 系列中的高性能、高配置的 Cortex-M3 内核 32 为处理器 STM32F103C8T6，72M 主频，LQFP48 封装，片内 FLASH 容量：64K，片内 SRAM 容量：20K。

STM32 家族主要产品系列家谱如下图 5-15 所示：

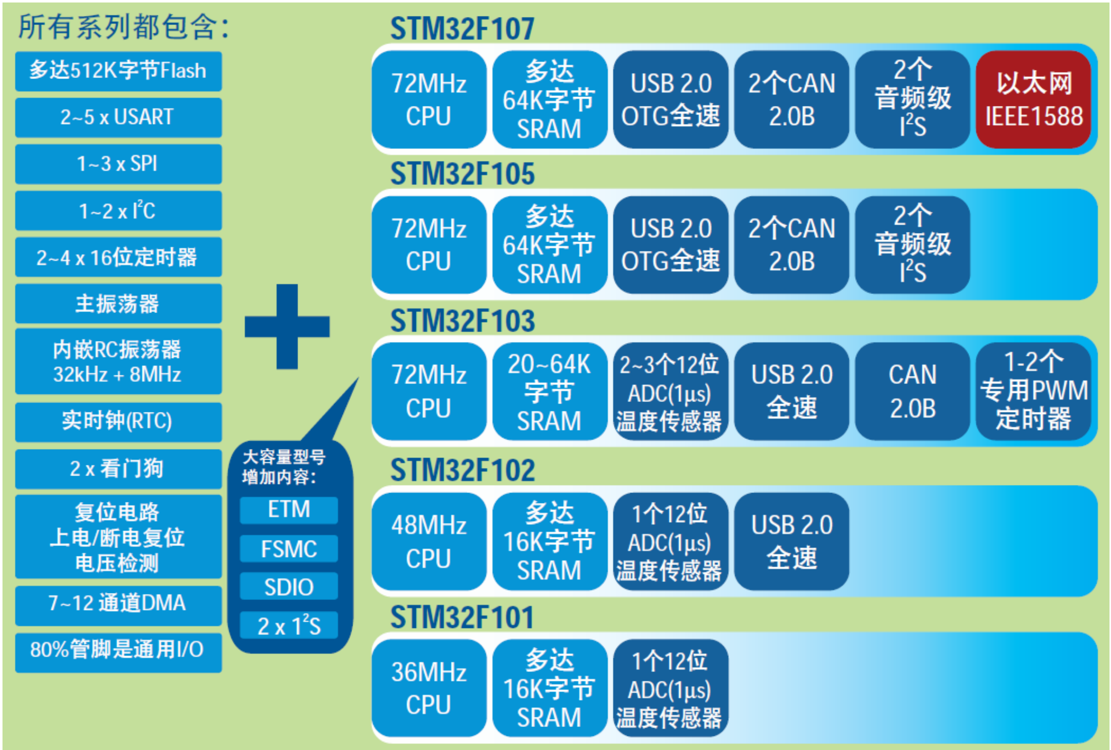


图 5-15 STM32 家族主要产品系列家谱

STM32F103 的产品列表下图 5-17 所示，核心板选用的是外设资源和管脚为 LQFP 封装的 STM32F103C8T6 芯片，该芯片具有 20K SRAM, 64K FLASH，3 个普通的 16 位定时器，1 个 16 位的高级定时器，2 个 SPI,2 个 IIC，3 个串口，1 个 USB，1 个 CAN，2 个 12 位的 ADC，37 个通用 IO 口，性价比非常高。STM32 产品选型如图 5-16 所示：

STM32(ARM Cortex-M3) 32位微控制器产品列表(截至2009年8月)

	型号	CPU频率(MHz)	程序空间(字节)	RAM(字节)	FSMC	定时器功能：16位普通(IC/OC/PWM)	定时器功能：16位高级(IC/OC/PWM)	定时器功能：16位基本	串行通信接口：SPI	串行通信接口：I²C	串行通信接口：USART/UART	串行通信接口：USB全速	串行通信接口：CAN 2.0	串行通信接口：以太网	串行通信接口：I²S	串行通信接口：SDIO	模拟端口：ADC(通道)	模拟端口：DAC(通道)	I/O端口	封装
48脚	STM32F103C4	72	16K	6K		2(8/8/8)	1(4/4/6)		1	1	2	1	1				2/(10)		37	LQFP48
	STM32F103C6	72	32K	10K		2(8/8/8)	1(4/4/6)		1	1	2	1	1				2/(10)		37	LQFP48
	STM32F103C8	72	64K	20K		3(12/12/12)	1(4/4/6)		2	2	3	1	1				2/(10)		37	LQFP48
	STM32F103CB	72	128K	20K		3(12/12/12)	1(4/4/6)		2	2	3	1	1				2/(10)		37	LQFP48
64脚	STM32F103R4	72	16K	6K		2(8/8/8)	1(4/4/6)		1	1	2	1	1				2/(16)		51	LQFP64/TFBGA64
	STM32F103R6	72	32K	10K		2(8/8/8)	1(4/4/6)		1	1	2	1	1				2/(16)		51	LQFP64/TFBGA64
	STM32F103R8	72	64K	20K		3(12/12/12)	1(4/4/6)		2	2	3	1	1				2/(16)		51	LQFP64/TFBGA64
	STM32F103RB	72	128K	20K		3(12/12/12)	1(4/4/6)		2	2	3	1	1				2/(16)		51	LQFP64/TFBGA64
	STM32F103RC	72	256K	48K		4(16/16/16)	2(8/8/12)	2	3	2	3+2	1	1		2	1	3/(16)	1(2)	51	LQFP64/WLCSP64
	STM32F103RD	72	384K	64K		4(16/16/16)	2(8/8/12)	2	3	2	3+2	1	1		2	1	3/(16)	1(2)	51	LQFP64/WLCSP64
	STM32F103RE	72	512K	64K		4(16/16/16)	2(8/8/12)	2	3	2	3+2	1	1		2	1	3/(16)	1(2)	51	LQFP64/WLCSP64
100脚	STM32F103V8	72	64K	20K		3(12/12/12)	1(4/4/6)		2	2	3	1	1				2/(16)		80	LQFP100/LFBGA100
	STM32F103VB	72	128K	20K		3(12/12/12)	1(4/4/6)		2	2	3	1	1				2/(16)		80	LQFP100/LFBGA100
	STM32F103VC	72	256K	48K	●	4(16/16/16)	2(8/8/12)	2	3	2	3+2	1	1		2	1	3/(16)	1(2)	80	LQFP100/LFBGA100
	STM32F103VD	72	384K	64K	●	4(16/16/16)	2(8/8/12)	2	3	2	3+2	1	1		2	1	3/(16)	1(2)	80	LQFP100/LFBGA100
	STM32F103VE	72	512K	64K	●	4(16/16/16)	2(8/8/12)	2	3	2	3+2	1	1		2	1	3/(16)	1(2)	80	LQFP100/LFBGA100

图 5-16 STM32F103 的产品列表

通过上表可以看到，STM32F103C8T6 暂时不提供 FSMC，以太网，I2S，SDIO,DAC 通道等接口，这些接口其他型号有提供，具体的大家可以参考上面这张表。

该芯片内部的各个外设模块的架构部署图如图 5-17 所示：

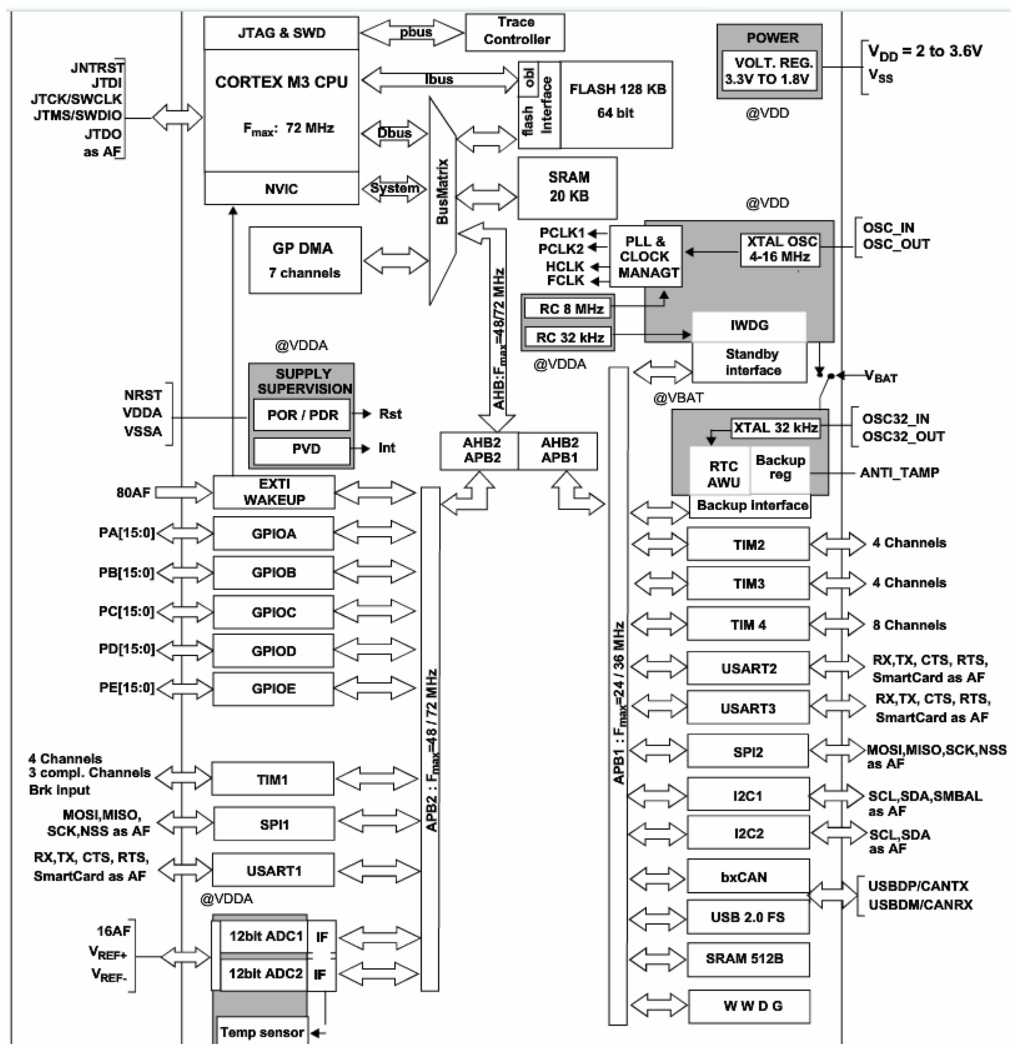


图 5-17 各外设模块架构部署图

5.2.4 LED指示灯

ARM 核心板提供了 2 路的 LED 指示灯，其中一个 LED 作为电源指示灯，如下图 5-18 所示，对应核心板上的【电源灯 DS2】，上电就会亮。另外的一个 LED 功能指示灯（在板子的标号是 DS1），当 PC13 输出低电平时点亮该 LED 灯。

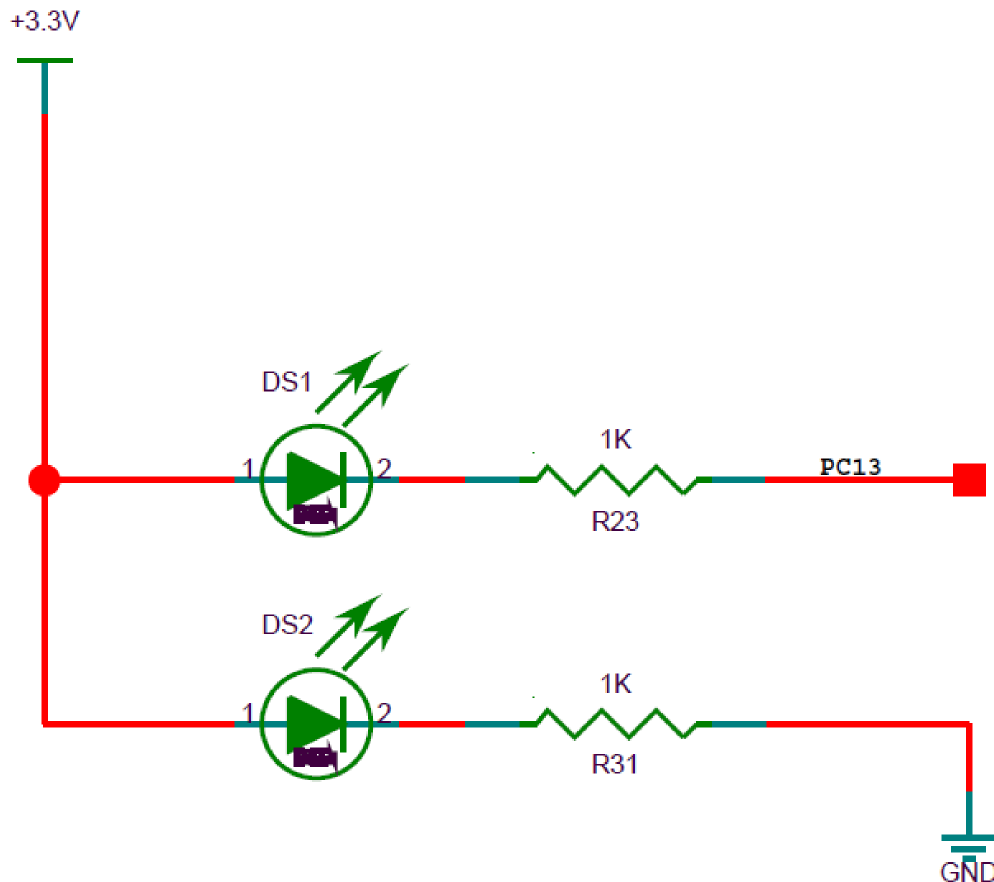


图 5-18 LED 电路图

图中的 DS1 受 PC13 控制，当 PC13 输出低电 平时点亮该 LED 灯，反之输出高电平时，LED 灯灭。

5.2.5 USART接口

核心板上提供了一组 USART 串口接口，如下图 5-19 所示，用户可以通过外接的信号线使用 USART 功能

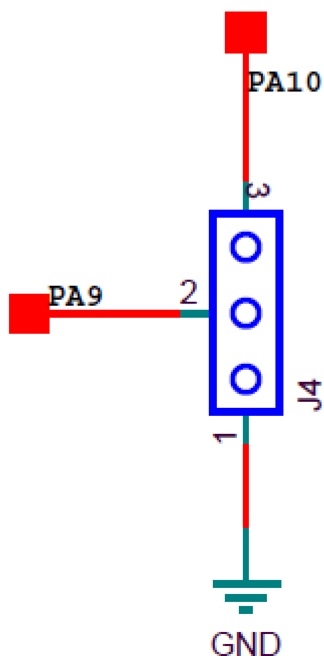


图 5-19　USART 串口接口

从原理图中可以看出，插针的 2 脚连接的是处理器的 PA9，也就是 USART 的 TX 端，3 脚连接的是处理器的 PA10，也就是 USART 的 RX 端，1 脚接的是 GND 端

5.2.6 复位系统

板子提供了一个复位按键，STM32F10XXX 是低电平复位的，STM32 的复位按键可用于复位整个核心板。

ARM 核心板设计采用的是最简单的“RC+按键”复位形式，复位电路的连接示意图如下图 5-20 所示，该复位电路可以实现上电复位功能和手动按键复位功能，下图就是复位按键的图：

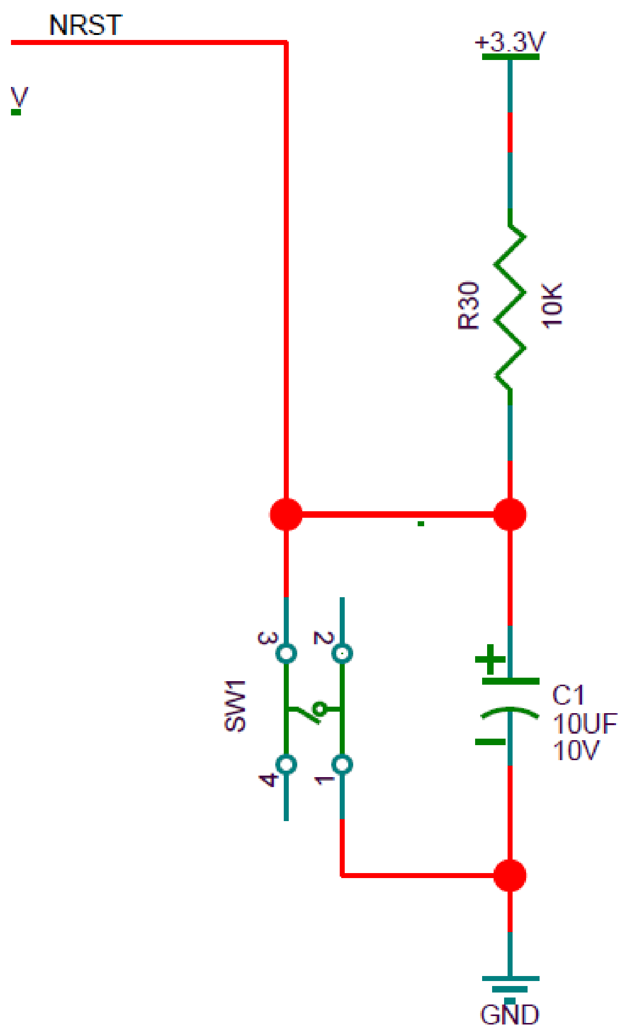


图 5-20 复位电路原理图

1）上电自动复位原理：上电钱电容里的电荷放光，系统上电瞬间，3.3V 通过电阻向电容充电，在充电过程中（充电过程中需要一些时间，而此时芯片始终处于复位状态下），电容的电压缓慢上升到 3.3V，只要电压没到 3.3V 这个值时芯片复位脚就近似于低电平，于是芯片检查到复位引脚低电平则导致系统复位，充电完毕后，电压接近 3.3V 时，芯片复位脚近似高电平，于是系统停止复位开始工作，复位完毕。

2）手动按键复位原理：系统正常工作时，电容里已经充满了电荷，所以电容的正极的电压为电源电压 3.3V，此时按下按键，则电容的正负极被短路，所以芯片复位脚为低电平，系统复位，同时由于电容正负极被短路电容里的电荷迅速释放干净，当按键释放时 3.3V 通过电阻向电容充电，原理与上电原理一致。

另外核心板上的一个功能按键是控制 J6 插针的 1 脚电平的，平时没按下去的时候它被上拉，处于高电平的状态，当 SW4 按键按下去的时候，J6 插针的 1 脚被拉到地，处于低电平的状态，如下图 5-21 所示，而 J6 又是控制 BOOT1 电压的，与 BOOT0 共同组成了核心板的启动模式选择。

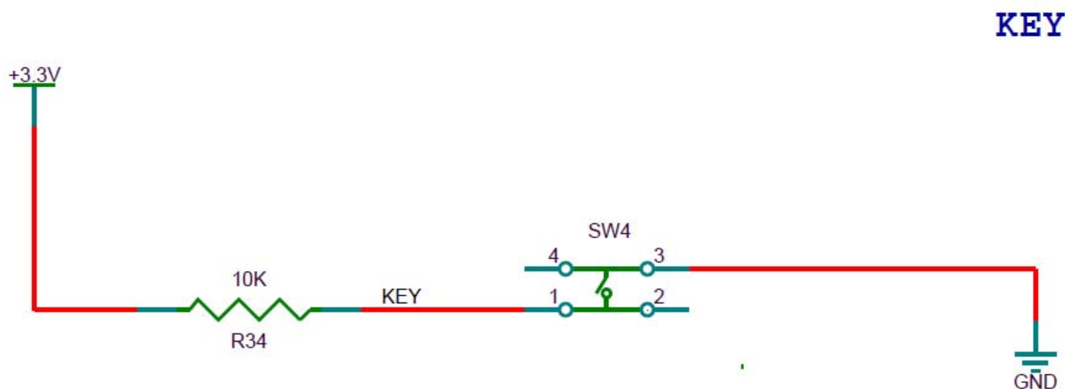


图 5-21 SW4 功能按键

5.2.7 标准的JTAG/SWD仿真调试下载接口

标准的 20 针 JTAG，直接可以和 J-LinkV8 仿真器连接的，同时支持 SWD（因为 STM32 支持 SWD）。可以用于调试 STM32，更方便的开发软件，如下图 5-22 所示，通过 USB 与仿真器相连接，再将仿真器的 JTAG 口与核心板的 JTAG 调试接口连接，就可以开始进行调试了：

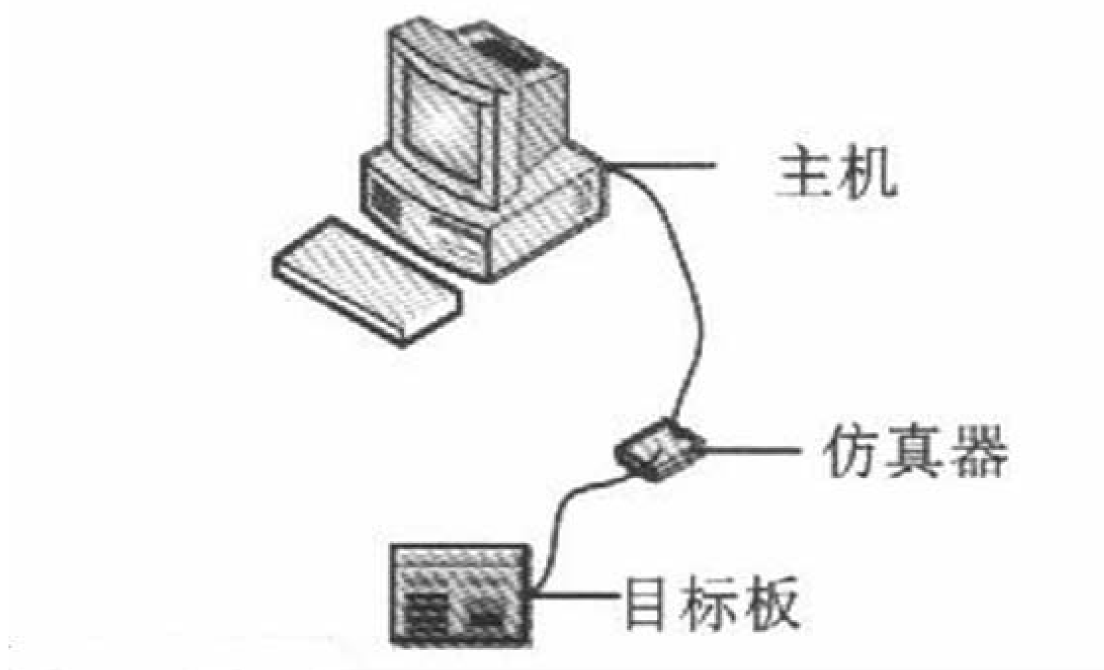


图 5-22 硬件连接

那么什么是处理器的调试接口呢？STM32F10XXX 使用 Cortex-M3 的内核，该内核含硬件 JTAG 调试模块，支持复杂的调试操作。硬件调试模块允许内核在取指（指令断点）或访问数据（数据断电）时停止。内核停止时，内核的内部状态和系统的外部状态都是可以查询的。完成查询后内核和外设可以被复原，程序将继续执行。

当 STM32F10X 微控制器连接到调试器并开始调试时，调试器将使用内核的硬件调试模

块进行调试操作。

支持两种调试接口：

- SW 串行接口
- JTAG 调试接口

下图 5-23 是我们 STM32 核心板 JTAG 调试接口原理图，我们可以通过该接口下载程序和硬件调试：

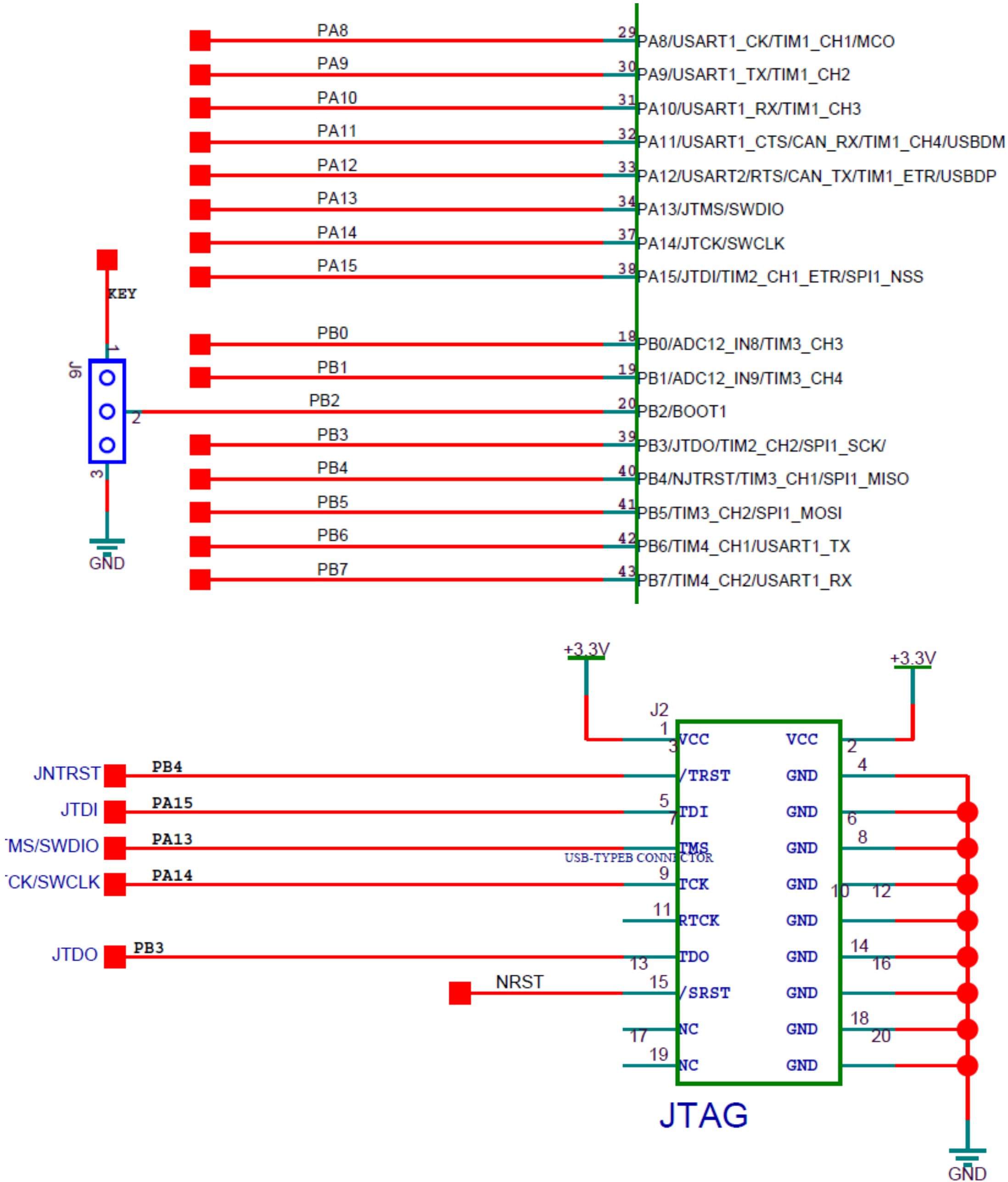


图 5-23 STM32 核心板 JTAG 调试接口原理图

调试接口的信号说明分布图如表 5-1 所示：

表 5-1 调试接口信号说明分布

SWJ-DP 端口引脚名称	JTAG 调试接口		SW 调试接口		引脚分频
	类型	描述	类型	调试功能	
JTMS/SWDIO	输入	JTAG 模式选择	输入/输出	串行数据输入/输出	PA13
JTCK/SWCLK	输入	JTAG 时钟	输入	串行时钟	PA14
JTDI	输入	JTAG 数据输入	--------	---------	PA15
JTDO/TRACESWO	输出	JTAG 数据输出	--------	跟 踪 时 为 TRACESWO 信号	PB3
JNTRST	输入	JTAG 模块复位	--------	----------	PB4

由上面这些接口可以知道，有了这个 JTAG 调试接口，就能够让我们更方便快捷的观察到处理器芯片的状态情况

5.2.8 USB全速接口

ARM 核心板带有一个 USB 2.0 全速从设备接口，如图 5-24 所示，可配置 1 到 8 个 USB 端点，它至少有 4 根信号线，包括 D+和 D-信号线，以及 Vcc 和 GND；当 USB 插入时（Vcc=5V）就会有下面几种情况：一种是如为充电器，则充电，这个没什么好说的；第二种是如为 USB 设备，则再分为低速和高速这两种情况，低速和高速是通过检测设备的 D+或 D-信号线是否有达到 3.5V 来区分的，因为插入前 D+和 D-都是低电平；

所谓“低速 USB 设备”是指当 USB 低速设备为（如鼠标，键盘）时，D-这条信号线为 3.5V 的电压，那么 STM32 芯片就会默认为是低速设备；反之，对于全速设备（如 U 盘，打印机，扫描仪），D+信号线就会被上拉到 3.5V

所以，当 USB 设备端插入主机后，主机通过检测 D+和 D-来确认是高速设备还是低速设备。

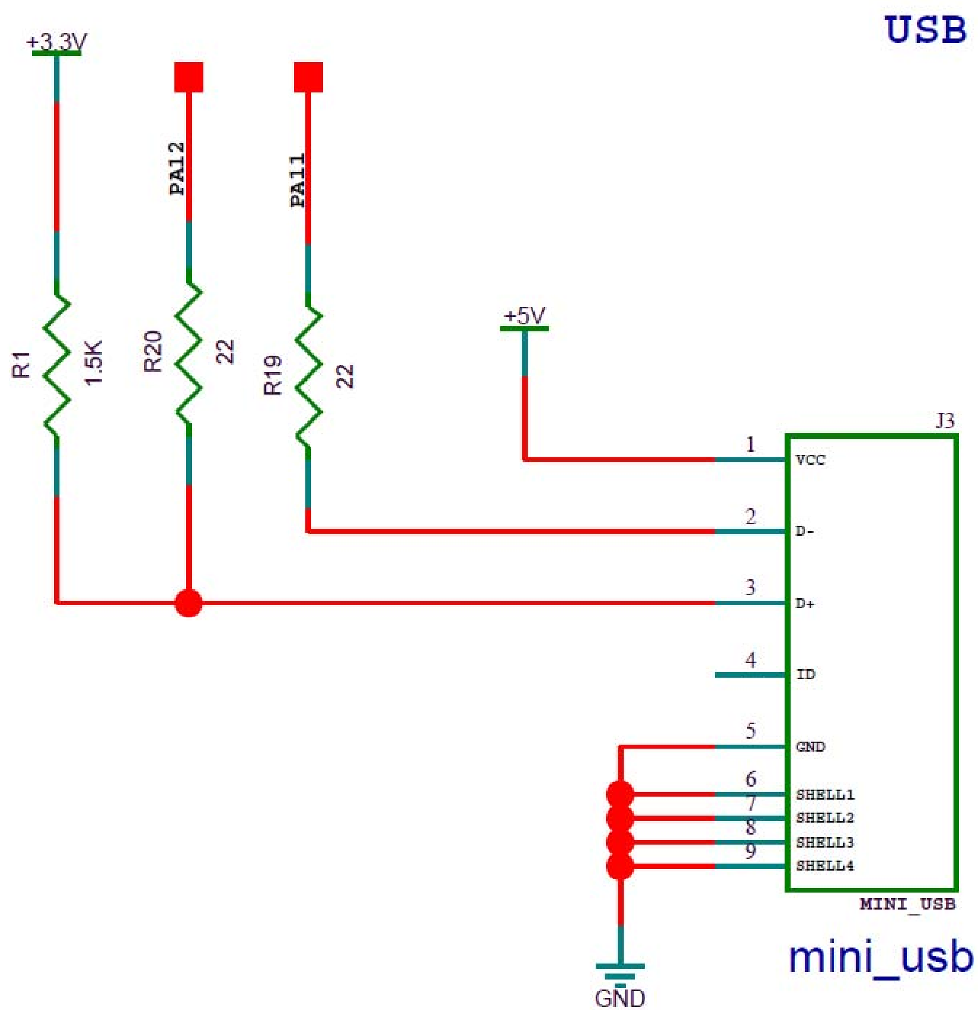


图 5-24　USB 2.0 全速从设备接口电路

5.2.9　连接器的说明

1）处理器启动方式的设置

下图 5-25 为 ARM 核心板处理器引导启动的条数布局图，实物图如图 5-26 所示：

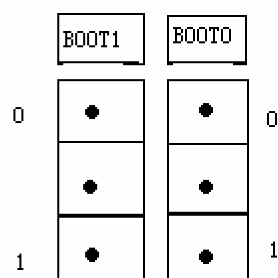


图 5-25 引导启动配置

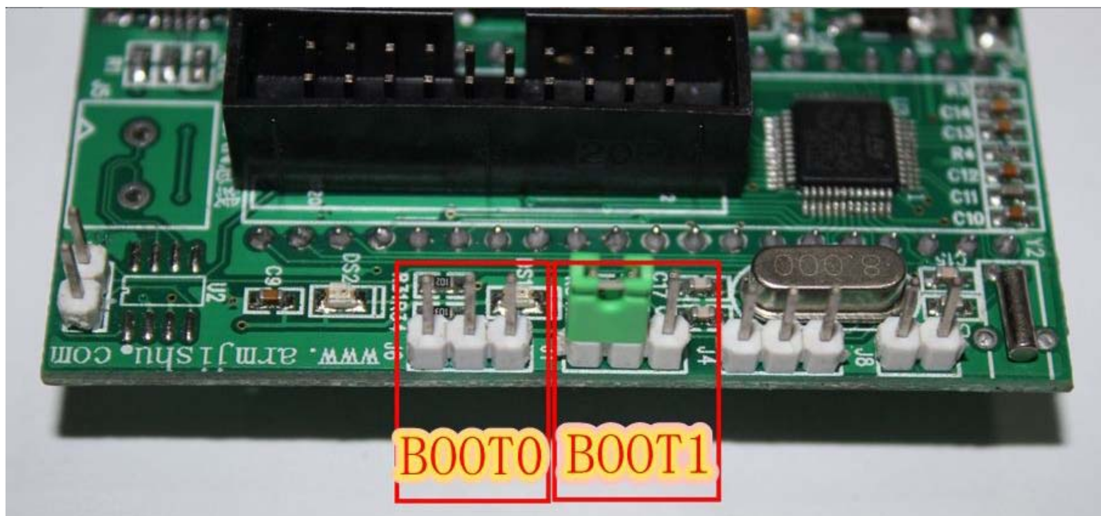


图 5-26 BOOT 启动模式接口实物图

上图中，上面的 BOOT0，BOOT1 用于设置 ARM 核心板的启动方式，其对应启动模式如下图表 5-2 所示：

表 5-2 BOOT 启动模式

BOOT1（J9）	BOOT0（J10）	功能	说明
ANY	0	User Boot(默认)	用主闪存存储器，即Flash启动
0	1	System Boot	系统存储器启动，用于串口下载
1	1	SRAM Boot	SRAM启动，用于SRAM中调试代码

从主闪存存储器启动：主闪存存储器被映射到启动空间（0x0000 0000），但仍然能够在它用来的地址（0x0800 0000）访问它，即闪存存储器的内容可以在两个地址区域访问，0x0000 0000 或 0x0800 0000。实现配置方法如图 5-27 所示：

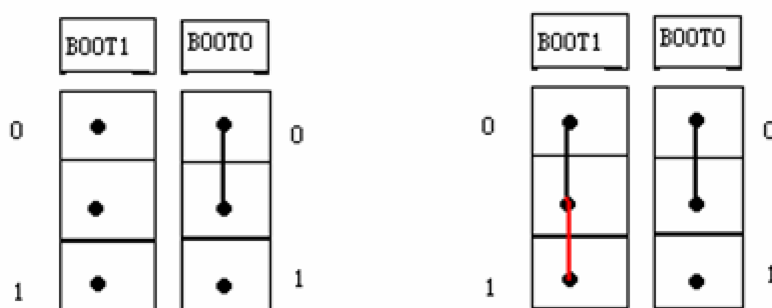


图 5-27 从主闪存存储器启动的 BOOT 配置

从系统存储器启动：系统存储器被映射到启动空间（0x0000 0000），但仍然能够在它原有的地址（0x1FFF F000）访问它。实现配置方法如图 5-28 所示：

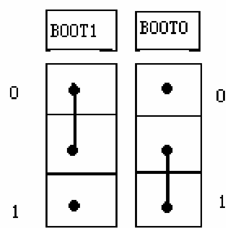


图 5-28 从系统存储器启动的 BOOT 配置

从内置 SRAM 启动：只能在 0x2000 0000 开始的地址区访问 SRAM。实现配置方法如图 5-29 所示：

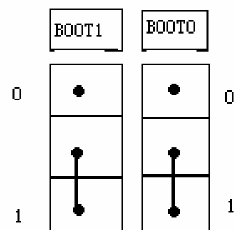


图 5-29 从内置 SRAM 启动的 BOOT 配置模式

2）串口功能接口

在核心板上还有一个串口功能的接口，为 3 个引脚的排针座子，分别是 TX、RX 和 GND，用户可以根据自己的需求直接连线，信号位置如下图 5-30 所示：

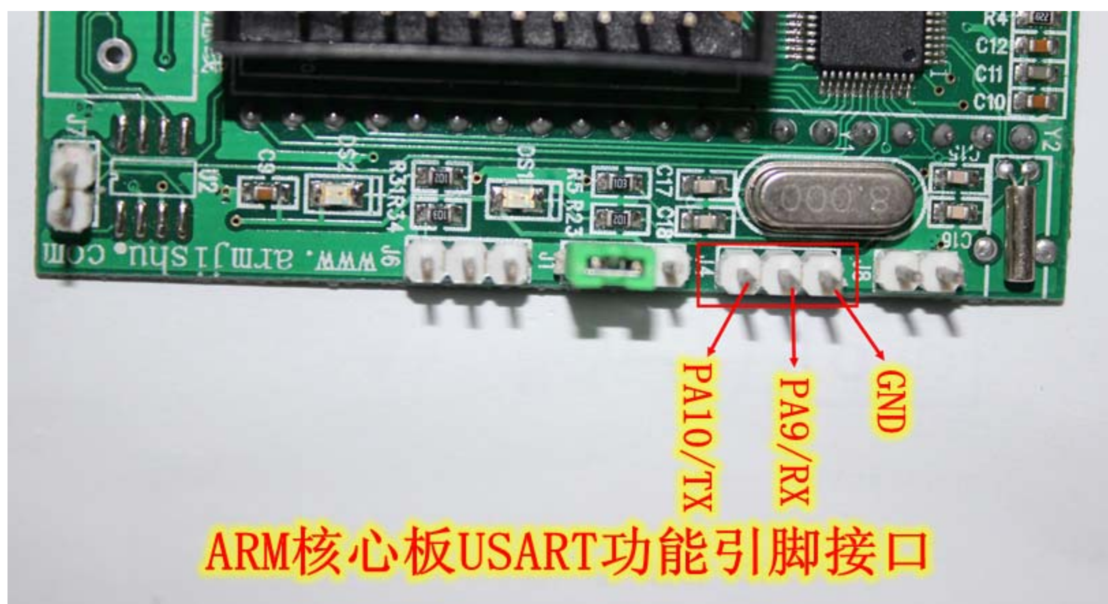


图 5-30 串口功能接口实物图

3）VBAT 功能接口

板子上还有一个 VBAT 的功能选择排针接口，只要把该接口短接上的话，就把芯片上的 VBAT 上拉到 3.3V，如下图 5-31 所示，只要短接 J8 接口，就把 VBAT 上拉到 3.3V

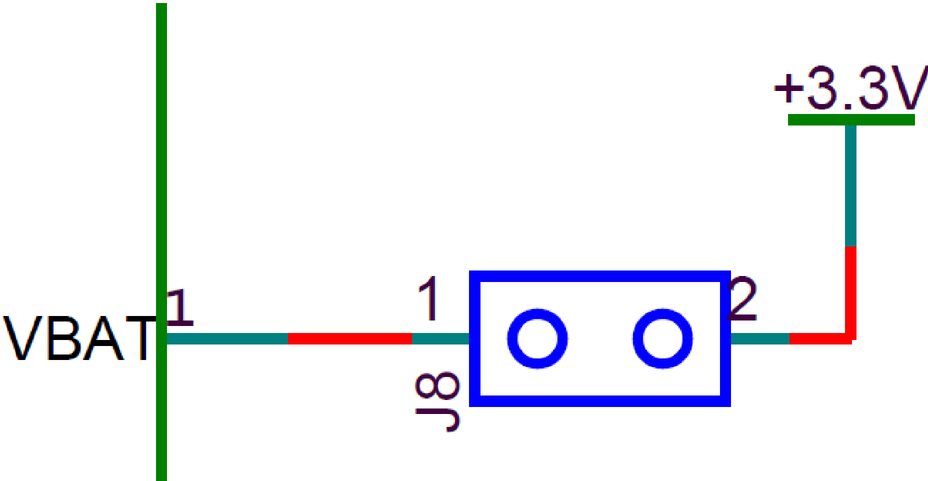


图 5-31 VBAT 的功能选择排针接口

5.3 通用输入/输出（GPIO）

5.3.1 管脚特性

103C8 处理器功能说明如图 5-32 所示：

STM32(ARM Cortex-M3) 32位微控制器产品列表(截至2009年8月)

	型号	CPU 频率 (MHz)	程序空间 (字节)	RAM (字节)	FSMC	定时器功能[1] 16位普通 (IC/OC/PWM)	定时器功能 16位高级 (IC/OC/PWM)	定时器功能 16位基本	串行通信接口 SPI	I²C	USART[2] UART	USB 全速	CAN 2.0	以太网	I²S	SDIO	模拟端口 ADC (通道)	模拟端口 DAC (通道)	I/O 端口	封装
48脚	STM32F103C4	72	16K	6K		2(8/8/8)	1(4/4/6)		1	1	2	1	1				2/(10)		37	LQFP48
	STM32F103C6	72	32K	10K		2(8/8/8)	1(4/4/6)		1	1	2	1	1				2/(10)		37	LQFP48
	STM32F103C8	72	64K	20K		3(12/12/12)	1(4/4/6)		2	2	3	1	1				2/(10)		37	LQFP48
	STM32F103CB	72	128K	20K		3(12/12/12)	1(4/4/6)		2	2	3	1	1				2/(10)		37	LQFP48

图 5-32 产品介绍

- STM32F103C8T6 总共有 36 个通用输入/输出（GPIO）口
- 每个 I/O 端口位可以自由编程
- I/O 端口寄存器可按 32 位字被访问（不允许半字或字节访问）

5.3.2 GPIO应用领域

- 通用 I/O 口
- 驱动 LED 或其他指示器
- 控制片外器件或片外器件通信
- 检测静态输入

5.3.3 管脚分配

STM32F103C8 处理器 GPIO 管脚描述如下表所示：

表 5-3：端口 A GPIO 管脚描述

管脚名称	类型	描 述
PA[15:0]	I/O	通用输入/输出 PA0 到 PA15

表 5-4：端口 B GPIO 管脚描述

管脚名称	类型	描 述
PB[15:0]	I/O	通用输入/输出 PB1 到 PB15

表 5-5：端口 C GPIO 管脚描述

管脚名称	类型	描 述
PC[15:13]	I/O	通用输入/输出 PC13 到 PC15 的 I/O 口功能有限制(同一时间内只有一个 I/O 口可以作为输出，速度必须限制在 2MHZ 内，而且这些 I/O 口不能当作电流源（如驱动 LED）)

表 5-6：端口 D GPIO 管脚描述

管脚名称	类型	描 述
PD[1:0]	I/O	通用输入/输出 PD0 到 PD1（该两个管脚被映射 OSC_IN 和 OSC_OUT 时钟）
NRST	I/O	复位

5.3.4 GPIO管脚内部硬件电路原理剖析

I/O 接口是一颗微控制器必须具备的最基本外设功能。通常在 ARM 里，所有 I/O 都是通用的，称为 GPIO（General Purpose Input/Output）。在 STM32 中，每个 GPIO 端口包含 16 个管脚，如 PA 端口是 PA0~PA15。GPIO 模块支持多种可编程输入/输出管脚，GPIO 模块包含以下特性：

1） 可编程控制 GPIO 中断

----- 包括屏蔽中断发生

----- 边沿触发（上升沿、下降沿、双边沿触发）

----- 电平触发（高电平触发、低电平触发）

2） 输入/输出管脚最大可承受 5V 电压

3） 可通过编程控制 GPIO 管脚配置：

----- 弱上拉或弱下拉电阻

----- 2mA、4mA、8mA 驱动，STM32 芯片管脚驱动最大是 25mA

GPIO 管脚可以被配置为多种工作模式，配置不同的模式实际就是内部的驱动电路有不一样，下面我们分析几种，大家可以从这里了解原理知识：

1. 高阻输入

高阻态是一个数字电路里常见的术语，指的是电路的一种输出状态，既不是高电平也不是低电平，如果高阻态再输入下一级电路的话，对下级电路无任何影响，和没接一样。

电路分析时高阻态可做开路理解。你可以把它看作输出（输入）电阻非常大。他的极限可以认为悬空。也就是说理论上高阻态不是悬空，它是对地或对电源电阻极大的状态。而实际应用上与引脚的悬空几乎是一样的，如图 5-33 所示：

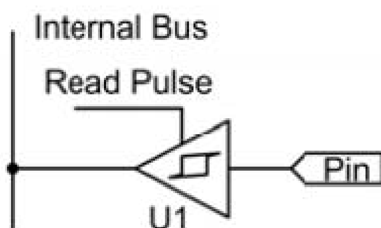


图 5-33 高阻输入模式

如上图所示，为 GPIO 管脚在高阻输入模式下的等效结构示意图，Pin 表示 GPIO 管脚；这是一个管脚的情况，其它管脚的结构也是同样的，输入模式的结构比较简单，就是一个带有施密特触发输入（Schmitt-triggered input）的三态缓冲器（U1），并具有很高的阻抗。施密特触发输入的作用是能将缓慢变化的或者是畸变的输入脉冲信号整形成比较理想的矩形脉冲信号。执行 GPIO 管脚读操作时，在读脉冲（Read Pulse）的作用下会把管脚（Pin）的当前电平状态读到内部总线上（Internal Bus）。

在不执行读操作时，它可以变成高阻抗的状态，使得外部管脚与内部总线之间是隔离的。

为什么会产生这种高阻抗的管脚设计呢？因为是很多管脚都连在同一根总线上，为了不干扰其他管脚，当某一个管脚在传送数据的适合，其他管脚配置成高阻抗，就不会干扰正在传送数据的管脚了，这样可以很多管脚同时共用一根总线，分时复用。

为减少信息传输线的数目，大多数计算机中的信息传输线采用总线形式，即凡要传输的同类信息都在同一组传输线，且信息是分时传送的。在计算机中一般有三组总线，即数据总线、地址总线和控制总线。为防止信息相互干扰，要求凡挂到总线上的寄存器或存储器等，它的输入输出端不仅能呈现 0、1 两个信息状态，而且还能产生一种高阻抗状态，即好像它们的输出被开关断开，对总线状态不起作用，此时总线可由其他器件占用。三态缓冲器即可实现上述功能，它除具有输入输出端之外，还有一控制端，就像一个开关一样，可以控制使其变成高阻抗状态。

2. 推挽输出

推挽输出可以提高输出功率，能够更好驱动外部的设备；推挽输出的原理：在功率放大器电路中大量采用推挽放大器电路，这种电路中用两只三极管构成一级放大器电路，两只三极管分别放大输入信号的正半周和负半周，即用一只三极管放大信号的正半周，用另一只三极管放大信号的负半周，两只三极管输出的半周信号在放大器负载上合并后得到一个完整周期的输出信号。

推挽放大器电路中，一只三极管工作在导通、放大状态时，另一只三极管处于截止状态，当输入信号变化到另一个半周后，原先导通、放大的三极管进入截止，而原先截止的三极管进入导通、放大状态，两只三极管在不断地交替导通放大和截止变化，所以称为推挽放大器。如图 5-35 所示

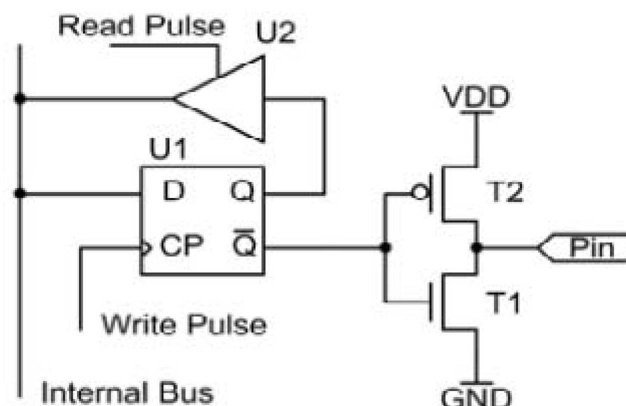


图 5-34　推挽输出模式

如上图所示，为 GPIO 管脚在推挽输出模式下的等效结构示意图。U1 是输出锁存器，执行 GPIO 管脚写操作时，在写脉冲（WritePulse）的作用下，数据被锁存到 Q 和/Q。T1 和 T2 构成 CMOS 反相器，T1 导通或 T2 导通时都表现出较低的阻抗，但 T1 和 T2 不会同时导通或同时关闭，最后形成的是推挽输出。在推挽输出模式下，GPIO 还具有回读功能，实现回读功能的是一个简单的三态门 U2。注意：执行回读功能时，读到的是管脚的输出锁存状态，而不是外部管脚 Pin 的状态。

推挽电路是两个参数相同的三极管或MOSFET,以推挽方式存在于电路中,各负责正负半周的波形放大任务,电路工作时，两只对称的功率开关管每次只有一个导通，所以导通损耗小、效率高。输出既可以向负载灌电流，也可以从负载抽取电流。推拉式输出级既提高电路的负载能力，又提高开关速度。

推挽放大器的输出级有两个“臂”(两组放大元件)，一个“臂”的电流增加时，另一个“臂”的电流则减小，二者的状态轮流转换。对负载而言，好像是 一个“臂”在推，一个“臂”在拉，共同完成电流输出任务。

3. 开漏输出

开漏输出就是不输出电压，低电平时接地，高电平时不接地。如果外接上拉电阻，则在输出高电平时电压会拉到上拉电阻的电源电压，如果开漏输出的管脚被上拉了，那么这个管脚将一直默认是输出高电平的。

一般来说，开漏是用来连接不同电平的器件，匹配电平用的，因为开漏引脚不连接外部的上拉电阻时，只能输出低电平，如果需要同时具备输出高电平的功能，则需要接上拉电阻，很好的一个优点是通过改变上拉电源的电压，便可以改变传输电平，比如加上上拉电阻就可以提供 TTL/CMOS 电平输出等。如图 5-35 所示

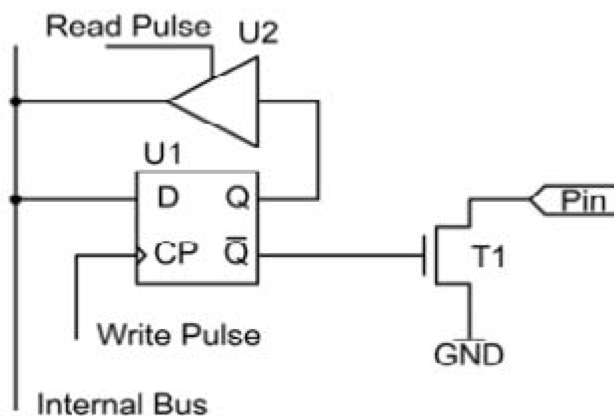


图 5-35 开漏输出模式

如上图所示，为 GPIO 管脚在开漏输出模式下的等效结构示意图。开漏输出和推挽输出相比结构基本相同，但只有下拉晶体管 T1 而没有上拉晶体管。同样，T1 实际上也是多组可编程选择的晶体管。开漏输出的实际作用就是一个开关，输出“1”时断开、输出“0”时连接到 GND（有一定内阻）。回读功能：读到的仍是输出锁存器的状态，而不是外部管脚 Pin 的状态。因此开漏输出模式是不能用来输入的。

开漏输出的优点是 IC 内部仅需很小的驱动电流就可以了，因为它主要是利用外部电路的驱动能力，这样可以减少 IC 内部的驱动，并且外部需要什么样的电压，就上拉到相应的电压，需要多大的电流，也可以通过改变上拉电阻来调节电流，所以开漏输出是非常灵活的一种输出。

4. 钳位二极管（用来保护 GPIO 管脚）

GPIO 内部具有钳位保护二极管，如下图 5-36 所示。其作用是防止从外部管脚 Pin 输入的电压过高或者过低。VDD 正常供电是 3.3V，如果从 Pin 输入的信号（假设任何输入信号都有一定的内阻）电压超过 VDD 加上二极管 D1 的导通压降（假定超过 VDD 电压 0.6V 的时候，那么二极管 D1 导通需要 0.6V 的压降），则二极管 D1 导通，这样就会把多于的电流引到 VDD，而真正输入到内部的信号电压不会超过 3.9V（3.3v+0.6v=3.9v）。同理，如果从 Pin 输入的信号电压比 GND 还低，则由于二极管 D2 的作用，会把实际输入内部的信号电压钳制在－0.6V 左右。

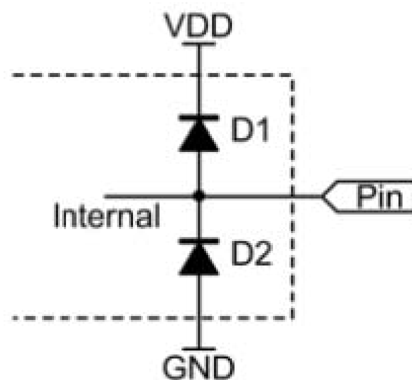


图 5-36 钳位保护二极管

假设 VDD＝3.3V，GPIO 设置在开漏模式下，外接 10kΩ上拉电阻连接到 5V 电源，

在输出“1”时，我们通过测量发现：GPIO 管脚上的电压并不会达到 5V，而是在 4V 上下，这正是内部钳位二极管在起作用。虽然输出电压达不到满幅的 5V，但对于实际的数字逻辑通常 3.5V 以上就算是高电平了。

如果确实想进一步提高输出电压，一种简单的做法是先在 GPIO 管脚上串联一只二极管（如 1N4148），然后再接上拉电阻。参见下图 5-37，框内是芯片内部电路。向管脚写“1”时，T1 关闭，在 Pin 处得到的电压是 3.3＋VD1＋VD3＝4.5V，电压提升效果明显；向管脚写“0”时，T1 导通，在 Pin 处得到的电压是 VD3＝0.6V，仍属低电平。

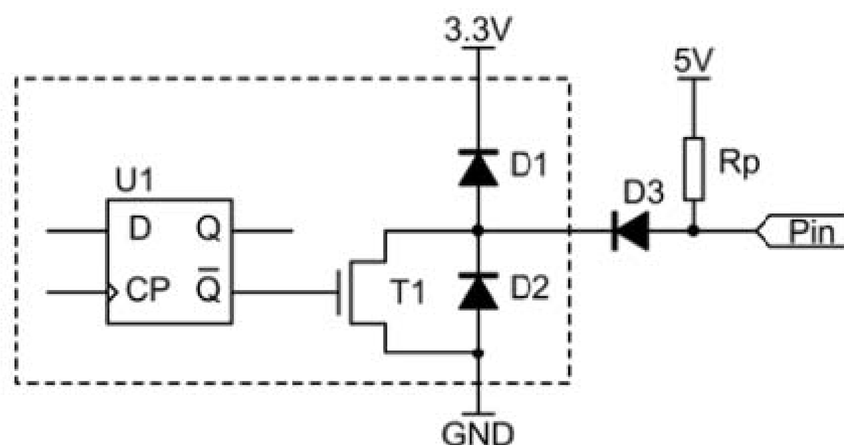


图 5-37 提高输出电压方法

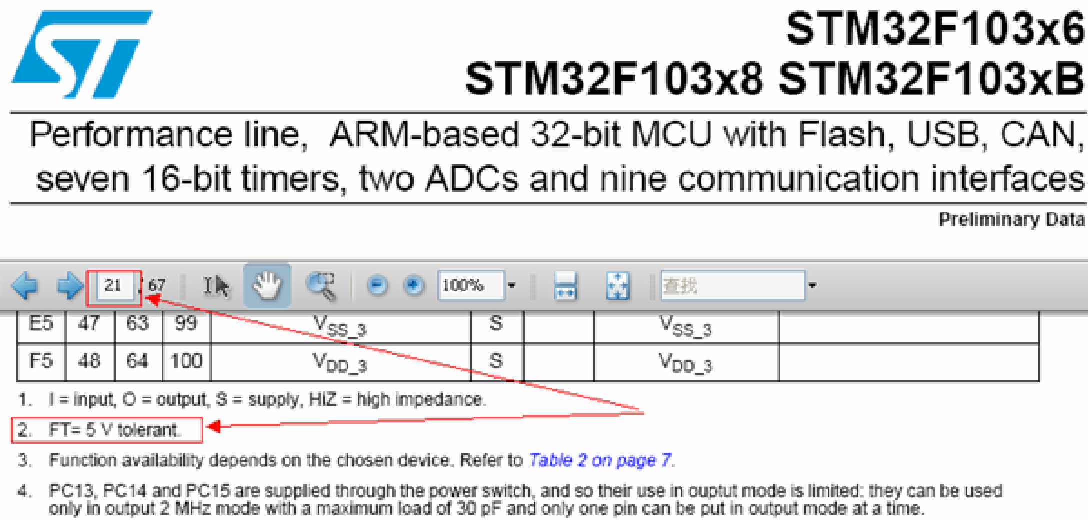
以上这节就是主要介绍芯片管脚的基本驱动原理，下面开始剖析具体的 STM32 芯片管脚。

5.3.5 STM32的GPIO管脚深入分析

上节我们介绍了芯片管脚实现的硬件原理，以及一个芯片管脚被配置成不同的模式实际是不同的驱动电路，这些驱动电路可以适用于不同的场合。STM23 的每个 GPIO 引脚都可以由软件配置成输出（推挽或开漏），输入（带或不带上拉或下拉）或复用的外设功能端口。多数 GPIO 引脚与数字或模拟的复用外设共用；除了具有模拟输入（ADC）功能的管脚之外，其他的 GPIO 引脚都有大电流通过能力，这些具体如下 8 种模式：

- 1）输入浮空（这个输入模式，输入电平必须由外部电路确定，要根据具体电路，加外部上拉电阻或下拉电阻，可以做按键识别）
- 2）输入上拉（打开 IO 内部上拉电阻）
- 3）输入下拉（打开 IO 内部下拉电阻）
- 4）模拟输入（应用 ADC 模拟输入）
- 5）开漏输出（输出端相当于三极管的集电极. 要得到高电平状态需要上拉电阻才行. 适合于做电流型的驱动,其吸收电流的能力相对强(一般 20ma 以内).能驱动大电流和大电压，LED 就使用这种模式。）
- 6）推挽式输出（可以输出高,低电平,连接数字器件。推挽式输出输出电阻小，带负载能力强）
- 7）推挽式复用功能（复用是指该引脚打开 remap 功能）
- 8）开漏复用功能（复用是指该引脚打开 remap 功能）

每个 IO 口可以自由编程，单 IO 口寄存器必须要按 32 位 bit 被访问。STM32 的很多 IO 口都是 5V 兼容的，这些 IO 口在与 5V 电平的外设连接的时候很有优势，具体哪些 IO 口是 5V 兼容的，可以从该芯片的数据手册管脚描述章节查到（I/O Level 标 FT 的就是 5V 电平兼容的）如图 5-38,我们打开 STM32F103C8T 的芯片手册，看到 21 页，对 FT 有详细的解释：



STM32F103x6
STM32F103x8 STM32F103xB

Performance line, ARM-based 32-bit MCU with Flash, USB, CAN, seven 16-bit timers, two ADCs and nine communication interfaces

Preliminary Data

E5	47	63	99	V_{SS_3}	S		V_{SS_3}	
F5	48	64	100	V_{DD_3}	S		V_{DD_3}	

1. I = input, O = output, S = supply, HiZ = high impedance.
2. FT= 5 V tolerant.
3. Function availability depends on the chosen device. Refer to *Table 2 on page 7*.
4. PC13, PC14 and PC15 are supplied through the power switch, and so their use in ouptut mode is limited: they can be used only in output 2 MHz mode with a maximum load of 30 pF and only one pin can be put in output mode at a time.

图 5-38 STM32F10x 芯片手册

STM32 的每个 IO 端口都有 7 个寄存器来控制（这里要注意的是关于芯片硬件管脚的信息可以通过芯片手册找到资料，有关控制某个管脚寄存器的说明需要参考《STM32F10xxx 参考手册》)；我们可以从手册截图的目录看到 7 个控制 GPIO 管脚的寄存器，如图 5-39 所示：

STM32F10xxx参考手册

参考手册

小，中和大容量的 STM32F101xx, STM32F102xx 和 STM32F103xx
ARM 内核 32 位高性能微控制器

目录 STM32F10xxx参考手册

图 5-39 GPIO 管脚寄存器

他们分别是两个端口配置寄存器，一个端口有 0~15 总共 16 个管脚；两个寄存器分别描

述输入和输出的，还有一个端口设置/清除寄存器来负责管脚是输出高电平还是低电平，一个端口清除寄存器和端口配置锁定寄存器。在需要的情况下，I/O 引脚的外设功能可以通过一个特定的操作锁定，以避免意外的写入 I/O 寄存器，这里我们仅介绍常用的几个寄存器，来完成我们的 LED 灯点灯实验的操作；在此，我们可以总结一下 STM32 的 IO 控制寄存器的作用：

1） STM32 的 CRL 和 CRH 寄存器主要是用来 **IO 管脚的方向和速率以及何种驱动模式**

2） STM32 的 ODR 寄存器是用来控制 **IO 口的输出高电平还是低电平**

3） STM32 的 IDR 寄存器主要是用来存储 **IO 口当前的输入状态（高低电平）的。**

4） STM32 的 BSRR 寄存器主要是用来直接对 IO 端某一位直接进行设置和清除操作，通过这个寄存器可以方便的直接修改一个引脚的高低电平

5） STM32 的 BRR 寄存器用来清除某端口的某一位位 0，如果该寄存器某位为 0，那么它所对应的那个引脚位不产生影响；如果该寄存器某位为 1，则清除对应的引脚位。

6） STM32 的 LCKR 用来锁定端口的配置，当对相应的端口位执行了 LOCK 序列后，在下次系统复位之前将不能再更改端口位的配置。

下面开始分析常用的两个 32 位配置寄存器 GPIOx_CRL 和 GPIOx_CRH，CRL 和 CRH 控制着每个 IO 口的模式及输出速率，我们下面来介绍一下两个寄存器，接下来我们看看端口低配置寄存器 CRL 的描述，偏移地址为：0x00,复位值为：0x4444 4444.如下表 5-7 和 5-8 所示：

表 5-7 端口低配置寄存器 CRL

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF7[1:0]		MODE7[1:0]		CNF6[1:0]		MODE6[1:0]		CNF5[1:0]		MODE5[1:0]		CNF4[1:0]		MODE4[1:0]	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNF3[1:0]		MODE3[1:0]		CNF2[1:0]		MODE2[1:0]		CNF1[1:0]		MODE1[1:0]		CNF0[1:0]		MODE0[1:0]	

表 5-8 端口低配置寄存器 CRL（功能说明）

位31:30	**CNFy[1:0]**：端口x配置位(y = 0…7)
27:26	软件通过这些位配置相应的I/O端口，请参考表15端口位配置表。
23:22	在输入模式(MODE[1:0]=00)：
19:18	00：模拟输入模式
15:14	01：浮空输入模式(复位后的状态)
11:10	10：上拉/下拉输入模式
7:6	11：保留
3:2	在输出模式(MODE[1:0]>00)：
	00：通用推挽输出模式
	01：通用开漏输出模式
	10：复用功能推挽输出模式
	11：复用功能开漏输出模式
位29:28	**MODEy[1:0]**：端口x的模式位(y = 0…7)
25:24	软件通过这些位配置相应的I/O端口，请参考表15端口位配置表。
21:20	00：输入模式(复位后的状态)

17:16	01：输出模式，最大速度10MHz
13:12	10：输出模式，最大速度2MHz
9:8, 5:4	11：输出模式，最大速度 50MHz
1:0	

该寄存器的复位值为 0X4444 4444（4 化成二进制为 0100），从上图可以看到，复位值其实就是配置端口为浮空输入模式。从上图还可以得出：STM32 的 CRL 控制着每个 IO 端口（A~G）的低 8 位的模式。每个 IO 端口的位占用 CRL 的 4 个位，高两位为 CNF，低两位为 MODE。这里我们可以记住几个常用的配置，比如 0X0 表示模拟输入模式（ADC 用）、0X3 表示推挽输出模式（做输出口用，50M 速率）、0X8 表示上/下拉输入模式（做输入口用）、0XB 表示复用输出（使用 IO 口的第二功能）。

STM32 的 IO 口位配置表如下表 5-9：

表 5-9 STM32 的 IO 口位配置表

<table>
<tr><th colspan="2">配置模式</th><th>CNF1</th><th>CNF0</th><th>MODE1</th><th>MODE0</th><th>PxODR 寄存器</th></tr>
<tr><td rowspan="2">通用输出</td><td>推挽式（Push-Pull）</td><td rowspan="2">0</td><td>0</td><td colspan="2" rowspan="4">01
10
11
见表 3.1.2</td><td>0 或 1</td></tr>
<tr><td>开漏（Open-Drain）</td><td>1</td><td>0 或 1</td></tr>
<tr><td rowspan="2">复用功能输出</td><td>推挽式（Push-Pull）</td><td rowspan="2">1</td><td>0</td><td>不使用</td></tr>
<tr><td>开漏（Open-Drain）</td><td>1</td><td>不使用</td></tr>
<tr><td rowspan="4">输入</td><td>模拟输入</td><td rowspan="2">0</td><td>0</td><td colspan="2" rowspan="4">00</td><td>不使用</td></tr>
<tr><td>浮空输入</td><td>1</td><td>不使用</td></tr>
<tr><td>下拉输入</td><td rowspan="2">1</td><td rowspan="2">0</td><td>0</td></tr>
<tr><td>上拉输入</td><td>1</td></tr>
</table>

可以看到 CNFX 是上面 CRL 寄存器里的配置位，这里配置位的不同，就会产生不同的 GPIO 管脚模式。

STM32 输出模式配置如下表 5-10 所示：

表 5-10　STM32 输出模式配置

MODE[1:0]	意义
00	保留
01	最大输出速度为 10MHz
10	最大输出速度为 2MHz
11	最大输出速度为 50MHz

这里 CRL 寄存器的 MODE 配置位的选项，不同的配置就会产生不同的速率。

CRH 的作用和 CRL 完全一样，只是 CRL 控制的是低 8 位输出口，而 CRH 控制的是高 8 位输出口，大家可以自己看 STM32 手册，我们在这里 CRH 就不做详细介绍了。

给个实例，比如我们要设置 PORTB 的 12 位为上拉输入，13 位为推挽输出。代码如下：

GPIOB->CRH&=0XFF00FFFF;//清掉这 2 个位原来的设置，同时也不影响其他位的设置

GPIOB->CRH|=0X00380000;　//PB12 输入，PB13 输出

GPIOB->ODR = 1<<12;//PB12 上拉（1 往右移 12 个位，从 PB0 开始，相当于把二进制 1 0000 0000 0000 赋值给 GPIOB_ODR，刚好对应了 PB12）

通过这 3 句话的配置，我们就设置了 PB12 为上拉输入，PB13 为推挽输出。

IDR 是一个 GPIOx_IDR 的端口输入数据寄存器的简称（ODR 是输入数据寄存器的简称），要想知道某个 IO 口的状态，就要读这个寄存器，再从读出的寄存器值分析出某个管脚位的状态，就可以知道这个管脚的状态了；IDR 寄存器只用了低 16 位。该寄存器为只读寄存器，并且只能以 16 位的形式读出。该寄存器各位的描述如下表 5-11 所示：

表 5-11 GPIOx_IDR 端口输入数据寄存器

31	30	29	28	27	26	25	24	23	22	21	20	19			
					18	17	16								
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR1 5	IDR14	IDR1 3	IDR1 2	IDR11	IDR10	IDR9	IDR8	IDR7	IDR 6	IDR 5	IRD 4	IDR3	IDR2	IDR1	IDR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

IDR 寄存器低 16 位配置功能如下表 5-12 所示

表 5-12　IDR 寄存器低 16 位配置功能

位 31:16	保留。始终读为 0。
位 15:0	IDRy[15:0]:端口输入数据（y=0…..15） 这些位为只读并只能以字（16 位）的形式读出。读出的值为对应 I/O 口的状态。

ODR 是一个端口输出数据寄存器，其作用就是控制端口的输出，对 ODR 对应寄存器位置 1 即对应的 GPIO 管脚就会输出高电平。该寄存器也只用了低 16 位，并且该寄存器可读可写，如果读的话，从该寄存器读出来的数据都是 0，所以读是没有意义的；只有写是有效的，该寄存器的各位描述如下表 5-13 所示：

表 5-13　GPIOx_ODR 端口输出数据寄存器

31	30	29	28	27	26	25	24	23	22	21	20	19			
					18	17	16								
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OD R15	ODR 14	ODR 13	OD R12	ODR 11	ODR 10	ODR 9	ODR 8	ODR 7	OD R6	OD R5	OD R4	ODR 3	ODR 2	ODR 1	ODR 0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

ODR 寄存器低 16 位配置功能如下表 5-14 所示

表 5-14 ODR 寄存器低 16 位配置功能

位 31: 16	保留。始终读为 0。
位 15:0	**ODRy[15:0]:**端口输入数据（y=0…..15） 这些位为只读并只能以字（16 位）的形式读出。读出的值为对应 I/O 口的状态。

GPIOx_IDR 是端口的输入数据寄存器，GPIOx_ODR 是端口的输出寄存器，我们配置引脚的输入输出模式是通过 GPIOx_CRL 和 GPIOx_CRH 两个寄存器来配置的，但是每个端口的 16 个引脚它们有的可能是输出模式，有的是输入模式，甚至一会输出一会输入，而

GPIOx_IDR 和 GPIOx_ODR 两个寄存器是以字模式（读一次就是访问 2 个字节，一个字等于 2 个字节）访问而不能以 bit 模式（bit 模式表示一次访问一个 bit 位，一个字节等于 8 个 bit，一个字等于 16 个 bit）访问，GPIOx_IDR 只能读，而 GPIOx_ODR 可以读写。

关于 GPIO 的输出模式下几种速度的区别：2MHz、10MHz、50MHz；这个又可以理解为输出驱动电路的不同响应速度（芯片内部在 I/O 口的输出部分安排了多个响应速度不同的输出驱动电路，用户可以根据自己的需要选择合适的驱动电路，通过选择速度来选择不同的输出驱动电路模块，达到最佳的噪声控制和降低功耗的目的）。

那为什么要几种速率呢？如果选择了不合适的速率会有什么影响呢？芯片管脚的速度就好比是信号收发的频率，速度快就表示信号收发的频率高；如果信号频率为 10MHz，而你配置了 2MHz 的带宽，那么就会丢失很多数据，很多数据点截取不到，这个 10MHz 的方波很可能就变成了正弦波。这个就好比是公路的设计时速，汽车速度低于设计时速时，可以平稳的运行，如果超过设计时速就会颠簸，甚至翻车。

所以芯片管脚的输入输出速率可以理解为，输出驱动电路的带宽，即一个驱动电路可以不失真地通过信号的最大频率；如果一个信号的频率超过了驱动电路的响应速度，就有可能信号失真。带宽速度高的驱动器耗电大、噪声也大，带宽低的驱动器耗电小、噪声也小；比如：高频的驱动电路，噪声也高，当不需要高的输出频率时，请选用低频驱动电路，这样非常有利于提高系统的 EMI 性能。当然如果要输出较高频率的信号，但却选用了较低频率的驱动模块，很可能会得到失真的输出信号。关键是 GPIO 的引脚速度跟应用匹配，比如：

1） USART 串口，若最大波特率只需 115.2kbps，1Mbps 速度等于 1000 kbps，那用 2MHz 的速度就够了，既省电也噪声小，STM32 最低的速率是 2MHz，已经可以满足要求了。
2） I2C 接口，若使用 400k 波特率，也可以选用 2M 的速度够了；当然若想把余量留大些，可以选用 10M 的 GPIO 引脚速度。
3） SPI 接口，若使用 18M 或 9M 波特率，需要选用 50M 的 GPIO 的引脚速度，我们这里一定要使得 GPIO 的速度大于外部应用的速度，这好像就是在高速上跑公交车，而不是在田埂上开赛车。高速上跑公交车，公交车可以开最快的速度，而不怕翻车。

这里提到了波特率，那么什么是波特率呢？波特率是指数据信号对载波的调制速率，它用单位时间内载波调制状态改变的次数来表示。波特率一次传输一个数据对象，而这个数据对象可能是几个比特（bit），所以波特率跟比特率是有区别的。

比特率在数字信道中，比特率是数字信号的传输速率，它用单位时间内传输的二进制代码的有效位(bit)数来表示，其单位为每秒比特数 bit/s(bps)、每秒千比特数(Kbps)或每秒兆比特数(Mbps)来表示(此处 k 和 m 分别为 1000bit/s 和 1000000bit/s)。

波特率与比特率的关系为：比特率 ＝ 波特率 X 单个调制状态对应的二进制位数。

5.3.6 在STM32中如何配置片内外设使用的IO端口

首先，一个外设在使用前，必须先配置和激活启动该外设的时钟，比如 GPIO 端口 B，那么就要激活 GPIOB 的时钟，比如 GPIOA，那么使用 PA2 管脚前，必须要前激活 GPIOA 端口的时钟，只有启动时钟后，这个外设才变得激活可用。

时钟被启动之后，再根据这个具体功能，对这个外设进行相应的设置和配置，这样的好处是可以降低 STM32 芯片的内部功耗，因为需要用到的外设才被激活，激活的外设会消耗芯片的比较多的功耗；不需要使用的外设无需初始化，这样设计可以降低芯片的功耗；好处尤其体现在类似手持设备，功耗比较小，使得电池更加耐用。

那么如何配置管脚采用哪种模式呢?这里粗略总结对应到外设的输入输出功能基本有三

种情况：

1）管脚输出：需要根据外围电路的配置选择对应的管脚为复用功能的推挽输出或复用功能的开漏输出。

2）管脚输入：则根据外围电路的配置可以选择浮空输入、带上拉输入或带下拉输入。

3）ADC 对应的管脚：配置管脚为模拟输入。

值得注意的是，这里如果把端口配置成复用输出功能，则该引脚与它当前连的信号电路断开，和复用功能信号电路连接，所以将管脚配置成复用输出功能后，如果只激活了该引脚的 GPIO 端口的时钟，而忘记把复用功能的时钟激活，那么它的输出将不确定，这样会产生异常的现象。

5.3.7 例程01 单个LED点灯闪烁程序

1. 示例简介

在神舟 51+ARM 开发板上，我们用一根杜邦线将 P0.0（PB8）和 JP19 上的 1 脚连起来。LED 灯的正极接的是 3.3V 电源，所以我们编程让 LED 负极拉低即 GPIO 引脚端口 B 的管脚 8 拉低，即 PB8 拉低，那么 LED 灯就会变亮，相关电路图如下图 5-40 所示：

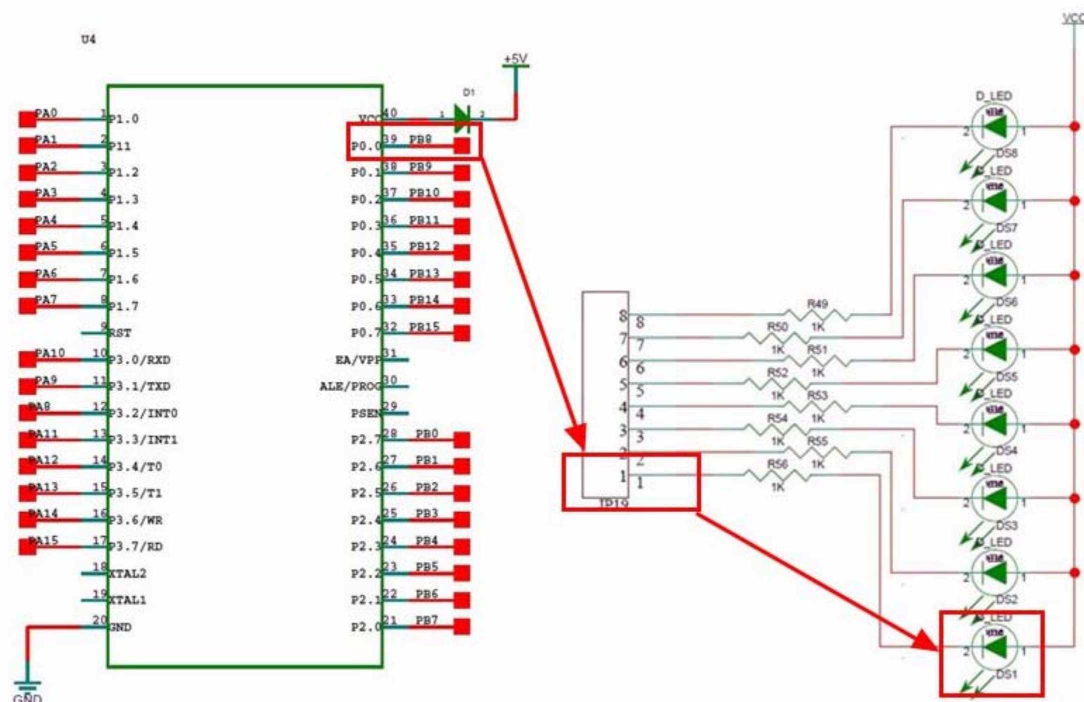


图 5-40 LED 相关实验电路原理图

注意到，这里采用GPIO管脚的低电平点灯，原因是：处理器的GPIO管脚只要输出低电平即可点灯，处理器功耗低；如果LED的一端固定接到GND地上，那么对于处理器的GPIO管脚点灯时，就必须输出高电平，这样增加处理器的功耗。同时，大家要注意，在设计LED灯限流时，串接的电阻放置的位置，有人会问，也可以放在LED的右边。一般，我们不会放在右边，主要是LED灯，人手可能会去触摸到，这样可能会将人体上的静电引导板件上，如果将电阻放在左边，静电经过电阻后，会消弱很多，以致不会一下子因为静电就将处理器烧毁。

一般的LED灯需要15~20毫安的电流才可以点亮，我们这里是3.3V的电压，经过1K欧姆

的限流电阻，用3.3V除以1000欧姆，理论值是33毫安的电流，足以点亮LED灯了。当然如果电流如果过大，就会使得经过的电流变下，LED灯可能点不亮或者比较暗；如果电阻过小，就会导致电流过大，可能烧掉LED灯，所以这个限流电阻选取也是有个范围的。

2. 调试说明

下载代码，并且按下【复位】键，在神舟 51+ARM 开发板上找到 LED 灯，可以看到该 LED 灯一亮一灭。

3. 关键代码：

相关代码如下图程序清单：

```
/************ www.armjishu.com *******/
int main(void)   //main 是程序入口
{
    /*  使能 APB2 总线的时钟，对 GPIO 的端口 B 时钟使能  */
    RCC->APB2ENR |= RCC_APB2Periph_GPIOB;

    /*-- GPIO Mode Configuration 速度，输入或输出  -----------------------*/
    /*-- GPIO CRL Configuration  设置 IO 端口低 8 位的模式（输入还是输出）---*/
    /*-- GPIO CRH Configuration  设置 IO 端口高 8 位的模式（输入还是输出）---*/
    GPIOB->CRH    &=    0xFFFFFFF0;
    GPIOB->CRH    |=    0x00000003;

    while (1)
    {
        GPIOB->BRR = GPIO_Pin_8;
        Delay(0x2FFFFF);
        GPIOB->BSRR = GPIO_Pin_8;
        Delay(0x2FFFFF);
    }
}
```

看原理图可以知道，因为 LED 的正极接的是 3.3V 电源端，所以当 PB8 管脚拉低成低电平的时候，LED 灯就会亮起来。

这里要注意的是在配置 STM32 外设的时候，任何时候都要先使能该外设的时钟！！！APB2ENR 寄存器是 APB2 总线上的外设时钟使能寄存器，其各位的描述如下表 5-15 所示：

表 5-15：寄存器 APB2ENR 各位描述

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADC3EN	USART1EN	TIM8EN	SPI1EN	TIM1EN	ADC2EN	ADC1EN	IOPGEN	IOPFEN	IOPEEN	IOPDEN	IOPCEN	IOPBEN	IOPAEN	保留	AFIOEN
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

我们要使能的是 PORTB 的时钟使能位，大家可以从上表看得到在 bit3 这个位，只需要将这个位置 1 就可以使能 PORTB 的时钟了，大家可以跟进去看看下面的这句代码，就能看到具体是设置的是这个 RCC 寄存器了

#define RCC_APB2Periph_GPIOB ((uint32_t)0x00000008);

这句代码 8 化成二进制是 1000 刚好是 bit3 的这个位，然后对这个 APB2ENR 的 GPIOB 端口时钟置位：

RCC->APB2ENR |= RCC_APB2Periph_GPIOB; /* 使能 APB2 总线的时钟，对 GPIO 的端口 B 时钟使能 */

这句代码相当于：RCC->APB2ENR |= 0x00000008；或上以后使得 APB2ENR 寄存器的第 bit3 这个位（从 0 开始，32 位寄存器，bit3 实际是在第 4 位）置 1，刚好是对应的 GPIO 端口 B 的时钟位，那么 GPIOB 的时钟就被使能了。

对 GPIOB->BRR = GPIO_Pin_8;这句代码来说，我们可以看到 GPIO_Pin_8 的定义如下：

#define GPIO_Pin_8 ((uint16_t)0x0100)

对 GPIOB_BRR 这个寄存器进行赋值，我们看下表 5-16：

表 5-16 GPIOB_BRR 寄存器

地址偏移：0x14															
复位值：0x0000 0000															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W

GPIOB_BRR 寄存器的各位功能配置如表 5-17

表 5-17 GPIOB_BRR 寄存器操作配置

位 31:16	保留。
位 15:0	**BRy**：清除端口 x 的位 y（y=0...15） 这些位只能写入并只能以字（16 位）的形式操作。 0：对应的 ODRy 位不产生影响 1：清除对应的 ODRy 位为 0

对 GPIOB_BRR 的 BR8 位进行置 1 操作，看寄存器说明可以知道，对 BR8 置 1 后，GPIOB_ODR 寄存器的 ODR8 位就为 0 了，使得 PB8 管脚输出低电平，灯被点亮（前面分析过，LED 灯是低电平点亮，具体原因请见前面的原理图分析）

同样，我们分析一下代码：GPIOB->BSRR = GPIO_Pin_8;下面看下 GPIOB_BSRR 寄存器，如表 5- 18 所示：

表 5-18 GPIOB_BSRR 寄存器

地址偏移：0x10															
复位值：0x0000 0000															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR 14	BR 13	BR 12	BR 11	BR 10	BR 9	BR 8	BR 7	BR 6	BR 5	BR 4	BR 3	BR 2	BR 1	BR 0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS 14	BS 13	BS 12	BS 11	BS 10	BS 9	BS 8	BS 7	BS 6	BS 5	BS 4	BS 3	BS 2	BS 1	BS 0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

GPIOB_BSRR 寄存器的各位功能配置如表 5-19

表 5-19　GPIOB_BSRR 寄存器操作配置

位	说明
位 31:16	**BRy**：清除端口 x 的位 y（y=0...15） 这些位只能写入并只能以字（16 位）的形式操作。 0：对对应的 ODRy 位不产生影响 1：清除对应的 ODRy 位为 0 注：如果同时设置了 BSy 和 BRy 的对应位，BSy 位起作用。
位 15:0	**BSy**：设置端口 x 的位 y（y=0...15） 这些位只能写入并只能以字（16 位）的形式操作。 0：对对应的 ODRy 位不产生影响 1：清除对应的 ODRy 位为 0

因为 GPIO_Pin_8 是等于 0x0100，所以 GPIOB_BSRR 寄存器的 BS8 被置 1，可以看到说明，该位置 1 后，使得 GPIOB_ODR 寄存器的 ODR8 位就为 1 了, PB8 管脚输出高电平，灯灭（前面分析过，LED 灯是低电平点亮，高电平熄灭，具体原因请见前面的原理图分析）。

其中 Delay(0x2FFFFF)是延时函数，所以在这个 while 循环里，LED 灯亮一段时间后，就熄灭一段时间，周而复始，交替进行。整段代码就都分析完了，至于代码中许多 define 的定义，例如：

```
/********* GPIOB 管脚的内存对应地址 *******/
#define PERIPH_BASE                  ((uint32_t)0x40000000)
#define APB2PERIPH_BASE              (PERIPH_BASE + 0x10000)
#define GPIOB_BASE                   (APB2PERIPH_BASE + 0x0c00)
#define GPIOB                    ((GPIO_TypeDef *) GPIOB_BASE)
```

可以参看前面的基础入门篇，如何将阅读寄存器的入门章节，这些都是于 STM32 芯片参考手册里的规定对应的。

更多关于这个寄存器的详细说明大家可以看《STM32F10xxx 参考手册》的第 7 章。

5.3.8 例程02　LED双灯闪烁实验

1. 示例简介

在神舟 51+ARM 开发板上，我们用两根杜邦线分别将 P0.0 和 JP19 的 1 脚、P0.1 和 JP19 的 2 脚连起来。LED 灯的正极接的是 3.3V 电源，所以我们编程让 LED 负极拉低即 GPIO 引脚端口 B 的 Pin8 和 Pin9 拉低，即 PB8 和 PB9；那么 LED 灯就会变亮；同样将 PB8 和 PB9 管脚拉高时，LED 就会灭掉；亮和灭各经过一段延时，就会变成闪烁的样子，这里我们编程增加了延时程序，相关电路图如下图 5-41 所示：

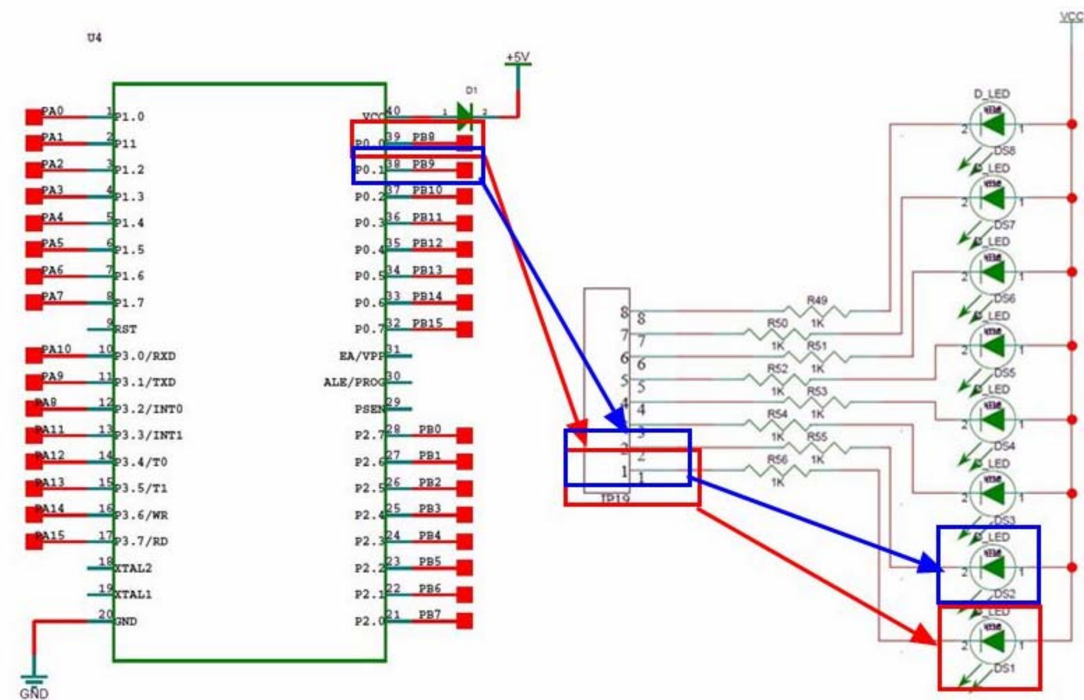


图 5-41 LED 实验相关原理图

2. 调试说明

下载代码，并且按下【复位】键，在神舟 51+ARM 开发板上可以看到 2 个 LED 灯不停的闪烁。

3. 关键代码

```
/************ www.armjishu.com *******/
int main(void)   //main 是程序入口
{
      /*  使能 APB2 总线的时钟，对 GPIO 的端口 B 时钟使能  */
      RCC->APB2ENR |= RCC_APB2Periph_GPIOB;

      /*-- GPIO Mode Configuration 速度，输入或输出  -----------------------*/
      /*-- GPIO CRL Configuration  设置 IO 端口低 8 位的模式（输入还是输出）---*/
      /*-- GPIO CRH Configuration  设置 IO 端口高 8 位的模式（输入还是输出）---*/
      GPIOB->CRH &=    0xFFFFFF00;
      GPIOB->CRH |=    0x00000033;    /*  设置 GPIOB 的 PB8 和 PB9 配置为通用推挽模式输出 50MHZ */

      while (1)
      {
       /*  对 BRR 设置为 1，则 GPIO 输出为 0 */
           GPIOB->BRR = GPIO_Pin_8;
           GPIOB->BRR = GPIO_Pin_9;
           Delay(0x2FFFFF);
```

```
        /* 对 BSRR 设置为 1，则 GPIO 输出为 1 */
            GPIOB->BSRR = GPIO_Pin_8;
            GPIOB->BSRR = GPIO_Pin_9;
            Delay(0x2FFFFF);
        }
    }
```

这里和上面程序不同之处是,使用的是 PB 端口同时控制 2 个管脚来控制 2 个 LED 的亮灭。

5.3.9 例程03 LED三个灯同时亮同时灭

1.示例简介

在神舟 51+ARM 开发板上，我们用三根杜邦线分别将 P0.0 和 JP19 的 1 脚、P0.1 和 JP19 的 2 脚、P1.0 和 JP19 的 3 脚连起来。LED 灯的正极接的是 3.3V 电源，所以我们编程让 PA0、PB8、PB9 三个管脚拉低；那么 LED 灯就会变亮；同样让 PA0、PB8、PB9 三个管脚拉高时，LED 就会灭掉；亮和灭各经过一段延时，就会变成闪烁的样子，这里我们编程增加了延时程序，相关电路图如下图 5-42 所示：

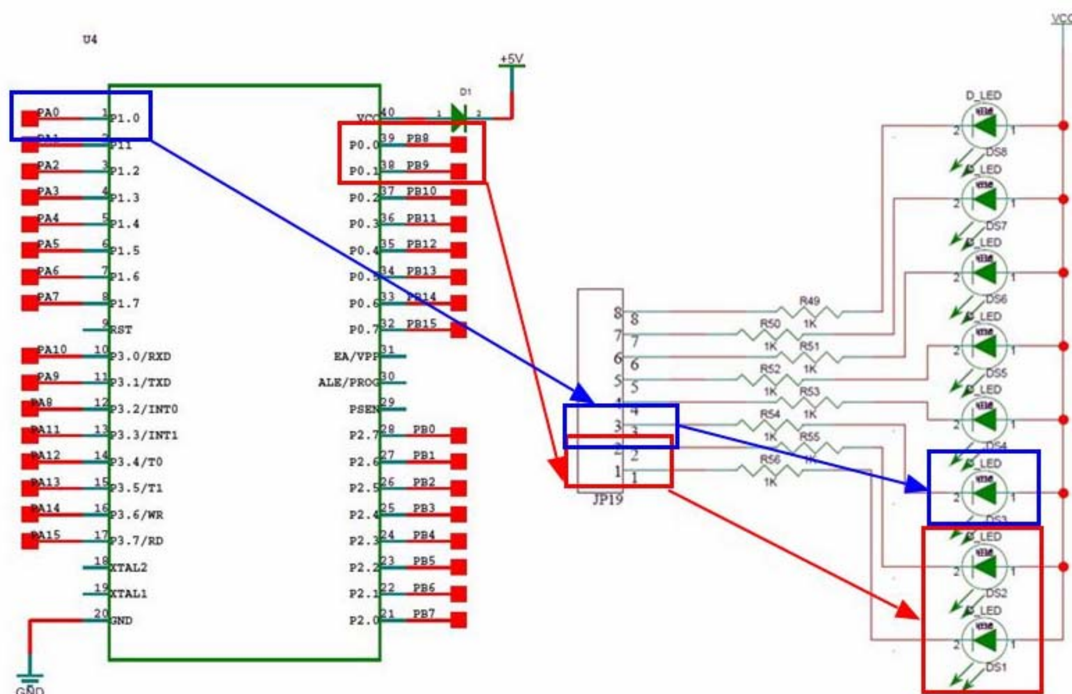


图 5-42 LED 实验相关原理图

2.调试说明

下载代码，并且按下【复位】键，在神舟 51+ARM 开发板上可以看到这 3 个 LED 灯不停的闪烁。

3.关键代码

```
/************ www.armjishu.com *******/
    int main(void)  //main 是程序入口
    {
        /*  使能 APB2 总线的时钟，对 GPIO 的端口 A 和 GPIO 的端口 B 时钟使能  */
        RCC->APB2ENR |= RCC_APB2Periph_GPIOA;
        RCC->APB2ENR |= RCC_APB2Periph_GPIOB;
        /*-- GPIO Mode Configuration 速度，输入或输出  -----------------------*/
        /*-- GPIO CRL Configuration  设置 IO 端口低 8 位的模式（输入还是输出）---*/
        /*-- GPIO CRH Configuration  设置 IO 端口高 8 位的模式（输入还是输出）---*/
        GPIOB->CRH &=   0xFFFFFF00;
        GPIOB->CRH |=   0x00000033;
                /*  设置 GPIOA 的 PA2 和 PA3 配置为通用推挽模式输出 50MHZ */
        GPIOA->CRL &=   0xFFFFFFF0;
        GPIOA->CRL |=   0x00000003;
                   /*  设置 GPIOB 的 PB2 配置为通用推挽模式输出 50MHZ */

        while (1)
        {
         /*  对 BRR 设置为 1，则 GPIO 输出为 0 */
            GPIOA->BRR = GPIO_Pin_0;
            GPIOB->BRR = GPIO_Pin_8;
            GPIOB->BRR = GPIO_Pin_9;
            Delay(0x2FFFFF);

        /*  对 BSRR 设置为 1，则 GPIO 输出为 1 */
            GPIOA->BSRR = GPIO_Pin_0;
            GPIOB->BSRR = GPIO_Pin_8;
            GPIOB->BSRR = GPIO_Pin_9;
            Delay(0x2FFFFF);
        }
    }
```

这里和上面程序不同之处是这里使用 PA0、PB8、PB9 三个管脚来同时控制三个 LED 灯的一亮一灭。

5.3.10例程04 LED流水灯程序

1. 示例简介

在神舟51+ARM开发板上，我们用8针排线将P0端口和JP19连起来，当GPIO管脚输出低电平时，对应的LED灯亮；当GPIO管脚输出高电平时，对应的LED灯灭。

图 5-43 为 LED 原理图，其中 GPIO 管脚上串的电阻，主要起限流作用。防止电流过大损坏 LED 和 GPIO 管脚：

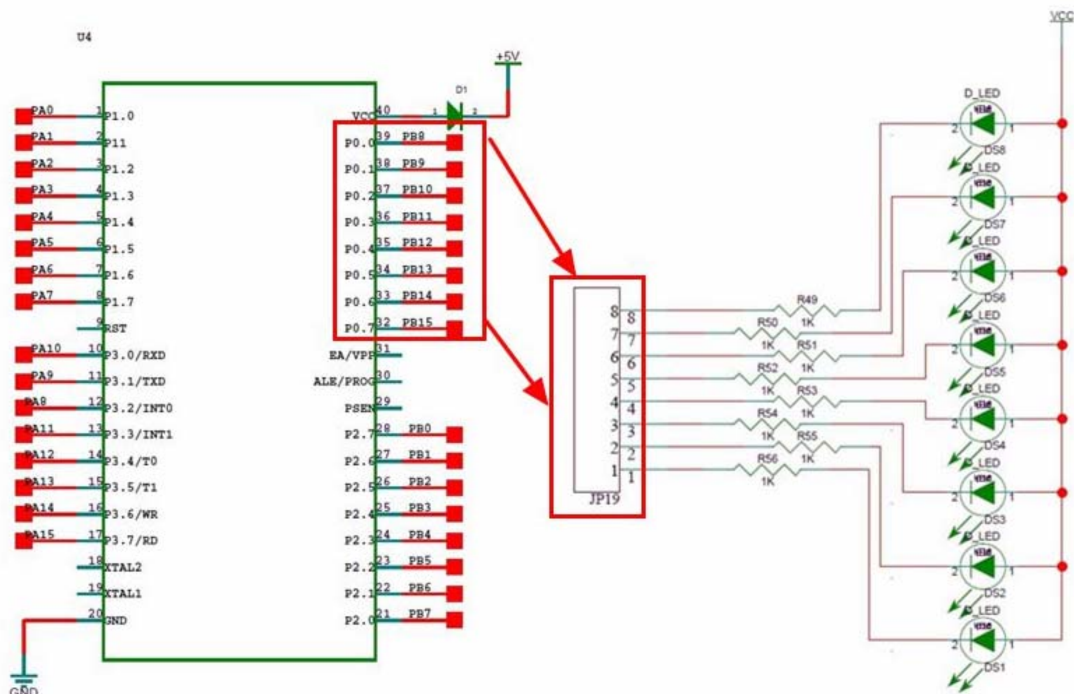


图 5-43 LED 实验相关原理图

2. 调试说明

下载代码，并且按下【复位】键，在神舟51+ARM板上，可以看到这8个灯轮流闪烁，流水灯。

3. 关键代码

```
/************ www.armjishu.com *******/
int main(void)   //main 是程序入口
{
    /*  使能 APB2 总线的时钟，对 GPIO 的端口 B 时钟使能  */
        RCC->APB2ENR |= RCC_APB2Periph_GPIOB;
    /*-- GPIO Mode Configuration 速度，输入或输出  -----------------------*/
    /*-- GPIO CRL Configuration  设置 IO 端口低 8 位的模式（输入还是输出）---*/
    /*-- GPIO CRH Configuration  设置 IO 端口高 8 位的模式（输入还是输出）---*/
    GPIOB->CRH &=    0x00000000;
    GPIOB->CRH |=    0x33333333;
                        /*  设置 GPIOB 的 PB8~BP15 配置为通用推挽模式输出 50MHZ */
    while (1)
    {
    /*  对 BSRR 设置为 1，则 GPIO 输出为 1,LED 灯灭  */
        GPIOB->BSRR = GPIO_Pin_8;            /* LED 灭*/
        Delay(0x5FFFF);                       /*  延时  */
        GPIOB->BSRR = GPIO_Pin_9;            /* LED 灭*/
        Delay(0x5FFFF);                       /*  延时  */
        GPIOB->BSRR = GPIO_Pin_10;           /* LED 灭*/
```

```
        Delay(0x5FFFF);                     /* 延时 */
        GPIOB->BSRR = GPIO_Pin_11;          /* LED 灭*/
        Delay(0x5FFFF);                     /* 延时 */
        GPIOB->BSRR = GPIO_Pin_12;          /* LED 灭*/
        Delay(0x5FFFF);                     /* 延时 */
        GPIOB->BSRR = GPIO_Pin_13;          /* LED 灭*/
        Delay(0x5FFFF);                     /* 延时 */
        GPIOB->BSRR = GPIO_Pin_14;          /* LED 灭*/
        Delay(0x5FFFF);                     /* 延时 */
        GPIOB->BSRR = GPIO_Pin_15;          /* LED 灭*/
        Delay(0x5FFFF);                     /* 延时 */

    /* 对 BRR 设置为 1，则 GPIO 输出为 0，LED 灯亮 */
        GPIOB->BRR = GPIO_Pin_8;            /* LED 亮*/
        Delay(0x5FFFF);                     /* 延时 */
        GPIOB->BRR = GPIO_Pin_9;            /* LED 亮*/
        Delay(0x5FFFF);                     /* 延时 */
        GPIOB->BRR = GPIO_Pin_10;           /* LED 亮*/
        Delay(0x5FFFF);                     /* 延时 */
        GPIOB->BRR = GPIO_Pin_11;           /* LED 亮*/
        Delay(0x5FFFF);                     /* 延时 */
        GPIOB->BRR = GPIO_Pin_12;           /* LED 亮*/
        Delay(0x5FFFF);                     /* 延时 */
        GPIOB->BRR = GPIO_Pin_13;           /* LED 亮*/
        Delay(0x5FFFF);                     /* 延时 */
        GPIOB->BRR = GPIO_Pin_14;           /* LED 亮*/
        Delay(0x5FFFF);                     /* 延时 */
        GPIOB->BRR = GPIO_Pin_15;           /* LED 亮*/
        Delay(0x5FFFF);                     /* 延时 */
    }
}
```

程序主要设计思路就是先将所有 LED 灯逐个经过延时后熄灭，然后再逐个被点亮，如此循环，形成 LED 流水灯。

5.4 时钟

5.4.1 什么是时钟

从 CPU 的时钟说起。

计算机是一个十分复杂的电子设备。它由各种集成电路和电子器件组成，每一块集成电路中都集成了数以万计的晶体管和其他电子元件。这样一个十分庞大的系统，要使它能够正常地工作，就必须有一个指挥，对各部分的工作进行协调。各个元件的动作就是在这个指挥下按不同的先后顺序完成自己的操作的，这个先后顺序我们称为时序。时序是计算机中一个非常重要的概念，如果时序出现错误，就会使系统发生故障，甚至造成死机。那么是谁来产

生和控制这个操作时序呢？这就是“时钟”。“时钟”可以认为是计算机的“心脏”，如同人一样，只有心脏在跳动，生命才能够继续。不要把计算机的“时钟”等同于普通的时钟，它实际上是由晶体振荡器产生的连续脉冲波，这些脉冲波的幅度和频率是不变的，这种时钟信号我们称为外部时钟。它们被送入 CPU 中，再形成 CPU 时钟。不同的 CPU，其外部时钟和 CPU 时钟的关系是不同的，下表列出了几种不同 CPU 外部时钟和 CPU 时钟的关系。

CPU 时钟周期通常为节拍脉冲或Ｔ周期，它是处理操作的最基本的单位。

在微程序控制器中，时序信号比较简单，一般采用节拍电位——节拍脉冲二级体制。就是说它只要一个节拍电位，在节拍电位又包含若干个节拍脉冲（时钟周期）。节拍电位表示一个ＣＰＵ周期的时间，而节拍脉冲把一个ＣＰＵ周期划分为几个叫较小的时间间隔。根据需要这些时间间隔可以相等，也可以不等。

指令周期是取出并执行一条指令的时间。

指令周期常常有若干个ＣＰＵ周期，ＣＰＵ周期也称为机器周期，由于ＣＰＵ访问一次内存所花费的时间较长，因此通常用内存中读取一个指令字的最短时间来规定ＣＰＵ周期。这就是说，这就是说一条指令取出阶段（通常为取指）需要一个ＣＰＵ周期时间。而一个ＣＰＵ周期时间又包含若干个时钟周期（通常为节拍脉冲或Ｔ周期，它是处理操作的最基本的单位）。这些时钟周期的总和则规定了一个ＣＰＵ周期的时间宽度。

5.4.2 STM32的时钟

系统时钟的选择是在启动时进行，复位时内部 8MHZ 的 RC 振荡器被选为默认的 CPU 时钟，随后可以选择外部的、具失效监控的 4-16MHZ 时钟；当检测到外部时钟失效时，它将被隔离，系统将自动地切换到内部的 RC 振荡器。

在 STM32 中，有五个时钟源，为 HSI、HSE、LSI、LSE、PLL，它们都是时钟所提供的来源：

1. HSI 是高速内部时钟，RC 振荡器，频率默认为 8MHz，时钟树的截图如下图 5-44 所示：

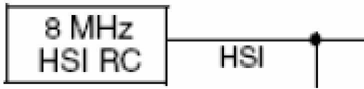


图 5-44 HIS 高速内部时钟

2. HSE 是高速外部时钟，可接石英/陶瓷谐振器，或者接外部时钟源，频率范围为 4MHz~16MHz，时钟树的截图如下图 5-45 所示：

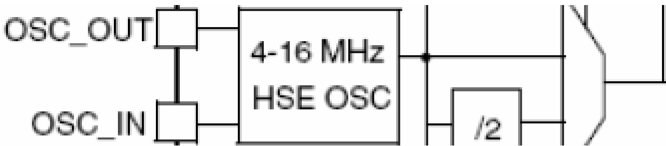


图 5-45 HSE 高速外部时钟

3. LSI 是低速内部时钟，RC 振荡器，频率为 40kHz，可以用于驱动独立看门狗和通过程序选择驱动 RTC（RTC 用于从停机/待机模式下自动唤醒系统），时钟树的截图如下图 5-46 所示：

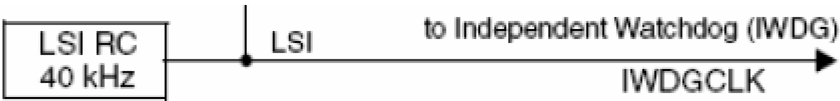


图 5-46 LSI 低速内部时钟

4. LSE 是低速外部时钟，接频率为 32.768kHz 的石英晶体，也可以被用来驱动 RTC，时钟树的截图如下图 5-47 所示：

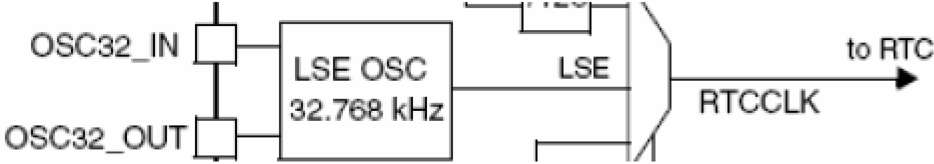


图 5-47 LSE 低速外部时钟

5. PLL 为锁相环倍频输出，其时钟输入源可选择为 HSI/2、HSE 或者 HSE/2。倍频可选择为 2~16 倍，但是其输出频率最大不得超过 72MHz，时钟树的截图如下图 5-48 所示：

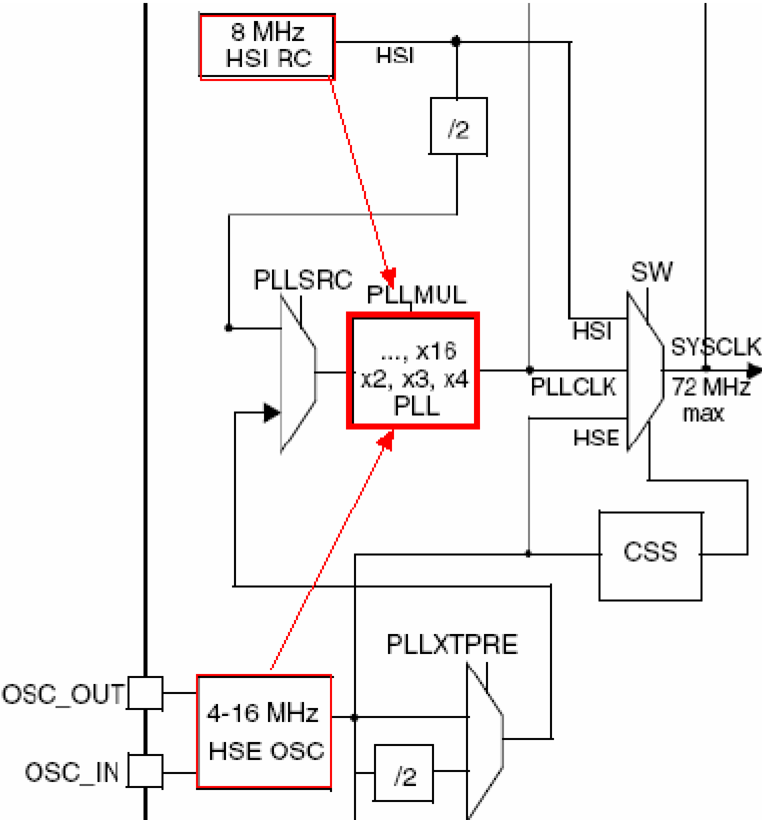


图 5-48 PLL 锁相环倍频输出

5.4.3 STM32的时钟深入分析

众所周知，微控制器（处理器）的运行必须要依赖周期性的时钟脉冲来驱动——往往由一个外部晶体振荡器提供时钟输入为始，最终转换为多个外部设备的周期性运作为末，这种时钟“能量”扩散流动的路径，犹如大树的养分通过主干流向各个分支，因此常称之为“时钟树”。在一些传统的低端 8 位单片机诸如 51，AVR，PIC 等单片机，其也具备自身的一个时钟树系统，但其中的绝大部分是不受用户控制的，亦即在单片机上电后，时钟树就固定在某种不可更改的状态(假设单片机处于正常工作的状态)。比如 51 单片机使用典型的 12MHz 晶振作为时钟源，则外设如 IO 口、定时器、串口等设备的驱动时钟速率便已经是固定的，用户无法将此时钟速率更改，除非更换晶振。

而 STM32 微控制器的时钟树则是可配置的，其时钟输入源与最终达到外设处的时钟速率不再有固定的关系，下面来详细解析 STM32 微控制器的时钟树。下图 5-50、5-51 是 STM32

微控制器的时钟树：

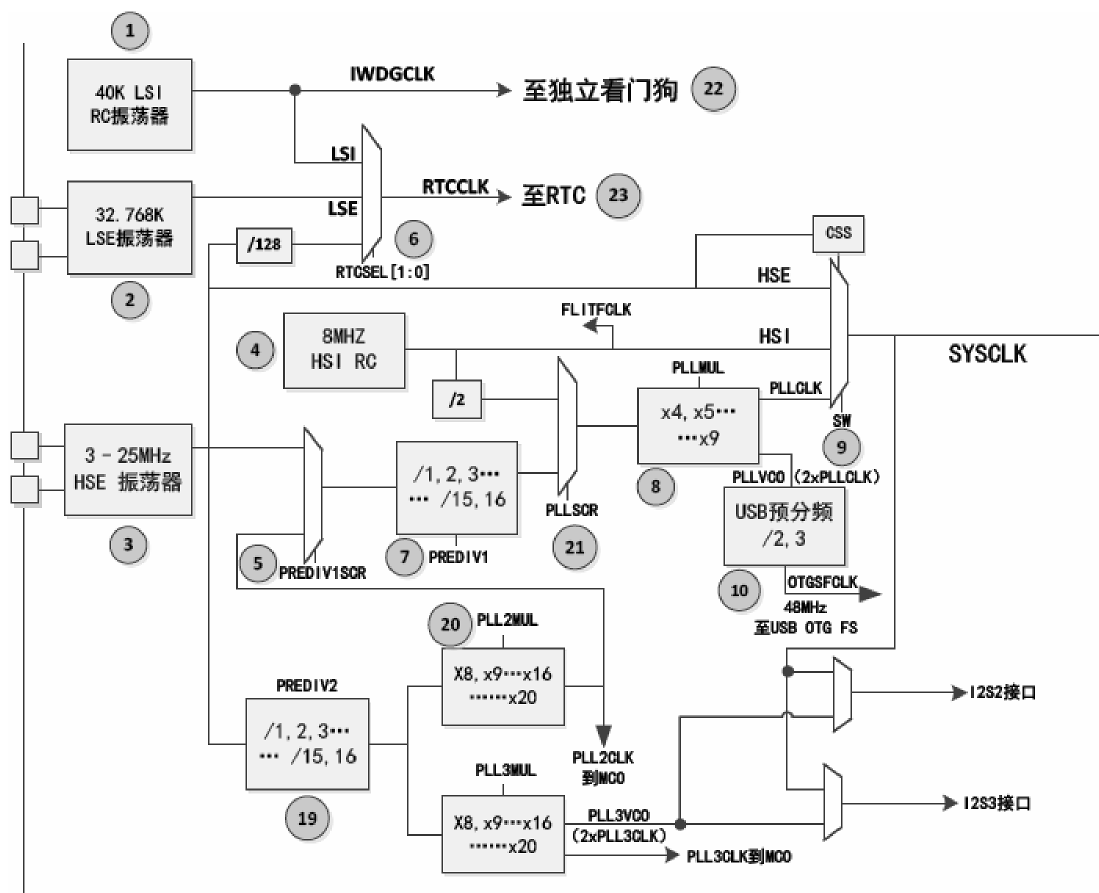


图 5-50 STM32 微控制器的时钟树

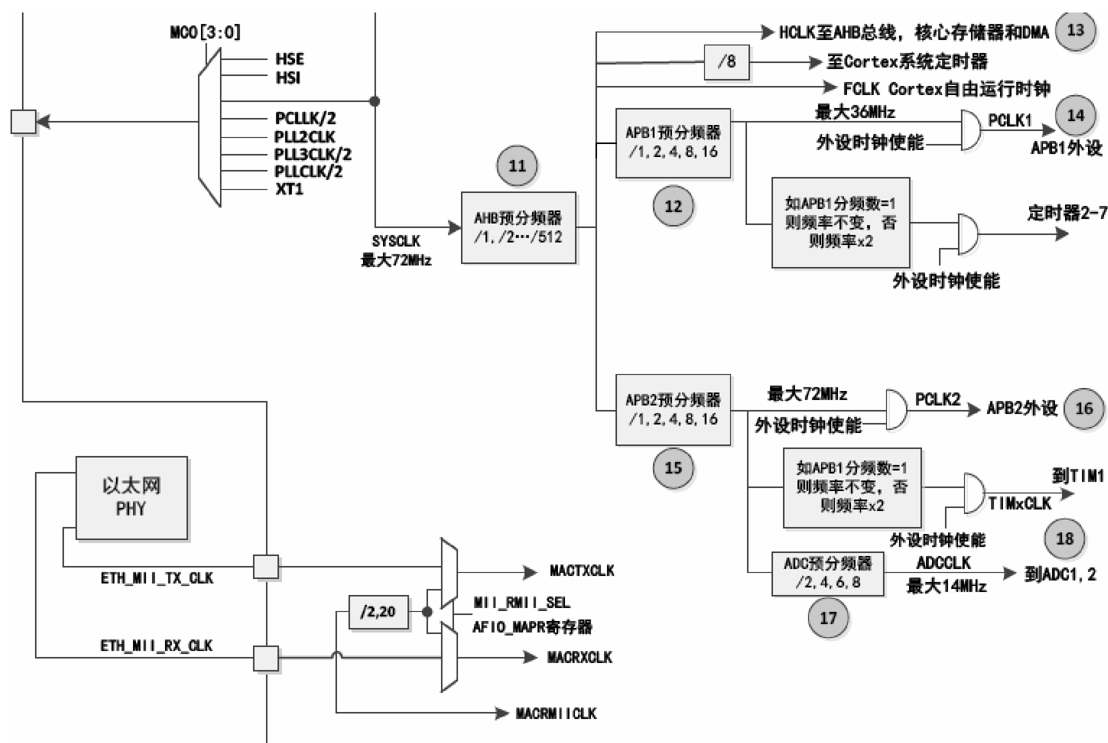


图 5-49 STM32 微控制器的时钟树

下表 5-20 是结合图表明出来的：

表 5-20 时钟树说明

标号	释义
1	内部低速振荡器（LSI，40Khz）
2	外部低速振荡器（LSE，32.768Khz）
3	外部高速振荡器（HSE，3-25MHz）
4	内部高速振荡器（HSI，8MHz）
5	PLL输入选择位
6	RTC时钟选择位
7	PLL1分频数寄存器
8	PLL1倍频寄存器
9	系统时钟选择位
10	USB分频寄存器
11	AHB分频寄存器
12	APB1分频寄存器
13	AHB总线
14	APB1外设总线
15	APB2分频寄存器
16	APB2外设总线
17	ADC预分频寄存器
18	ADC外设
19	PLL2分频数寄存器
20	PLL2倍频寄存器
21	PLL 时钟源选择寄存器
22	独立看门狗设备
23	RTC 设备

在认识这颗时钟树之前，首先要明确“主干”和最终的“分支”。假设使用外部 8MHz 晶振作为 STM32 的时钟输入源（这也是最常见的一种做法），则这个 8MHz 便是“主干”，而“分支”很显然是最终的外部设备比如通用输入输出设备（GPIO）。这样可以轻易找出第一条时钟的“脉络”：

3——5——7——21——8——9——11——13

对此条时钟路径做如下解析：

对于 3，首先是外部的 3-25MHz（前文已假设为 8MHz）输入；

对于 5，通过 PLL 选择位预先选择后续 PLL 分支的输入时钟（假设选择外部晶振）；

对于 7，设置外部晶振的分频数（假设 1 分频）；

对于 21，选择 PLL 倍频的时钟源（假设选择经过分频后的外部晶振时钟）；

对于 8，设置 PLL 倍频数（假设 9 倍频）；

对于 9，选择系统时钟源（假设选择经过 PLL 倍频所输出的时钟）；

对于 11，设置 AHB 总线分频数（假设 1 分频）；

对于 13，时钟到达 AHB 总线；

在上一章节中所介绍的 GPIO 外设属于 APB2 设备，即 GPIO 的时钟来源于 APB2 总线，同样在上图中也可以寻获 GPIO 外设的时钟轨迹：

3——5——7——21——8——9——11——15——16

对于 3，首先是外部的 3-25MHz（前文已假设为 8MHz）输入；

对于 5， 通过 PLL 选择位预先选择后续 PLL 分支的输入时钟（假设选择外部晶振）；

对于 7，设置外部晶振的分频数（假设 1 分频）；

对于 21，选择 PLL 倍频的时钟源（假设选择经过分频后的外部晶振时钟）；

对于 8，设置 PLL 倍频数（假设 9 倍频）；

对于 9，选择系统时钟源（假设选择经过 PLL 倍频所输出的时钟）；

对于 11，设置 AHB 总线分频数（假设 1 分频）；

对于 15，设置 APB2 总线分频数（假设 1 分频）

对于 16，时钟到达 APB2 总线；

现在来计算一下 GPIO 设备的最大驱动时钟速率（各个条件已在上述要点中假设）：

1) 由 3 所知晶振输入为 8MHz，由○5——○21 知 PLL 的时钟源为经过分频后的外部晶振时钟，并且此分频数为 1 分频，因此首先得出 PLL 的时钟源为：8MHz / 1 = 8MHz。

2) 由 8、9 知 PLL 倍频 9，且将 PLL 倍频后的时钟输出选择为系统时钟，则得出系统时钟为 8MHz * 9 = 72MHz。

3) 时钟到达 AHB 预分频器，由 11 知时钟经过 AHB 预分频器之后的速率仍为 72MHz。

4) 时钟到达 APB2 预分频器，由 15 经过 APB2 预分频器后速率仍为 72MHz。

5) 时钟到达 APB2 总线外设

上面是原理的剖析，如果再不明白的，可以接下来看例程代码，理论联系实践是最好的老师。

5.4.4 例程01 STM32芯片32MHZ频率下跑点灯程序

1. 示例简介

我们先用一根杜邦线将 P0.0（PB8）和 JP19 上的 1 脚连起来(其实，连 JP19 上的任意一脚都可)。点灯程序在时钟主频 32MHz 下面运行，LED 灯的正极接的是 3.3V 电源，所以我们编程让 LED 负极拉低即 GPIO 引脚端口 B 的管脚 8 拉低，即 PB8 拉低，那么 LED 灯就会变亮，相关电路图如下图 5-50 所示：

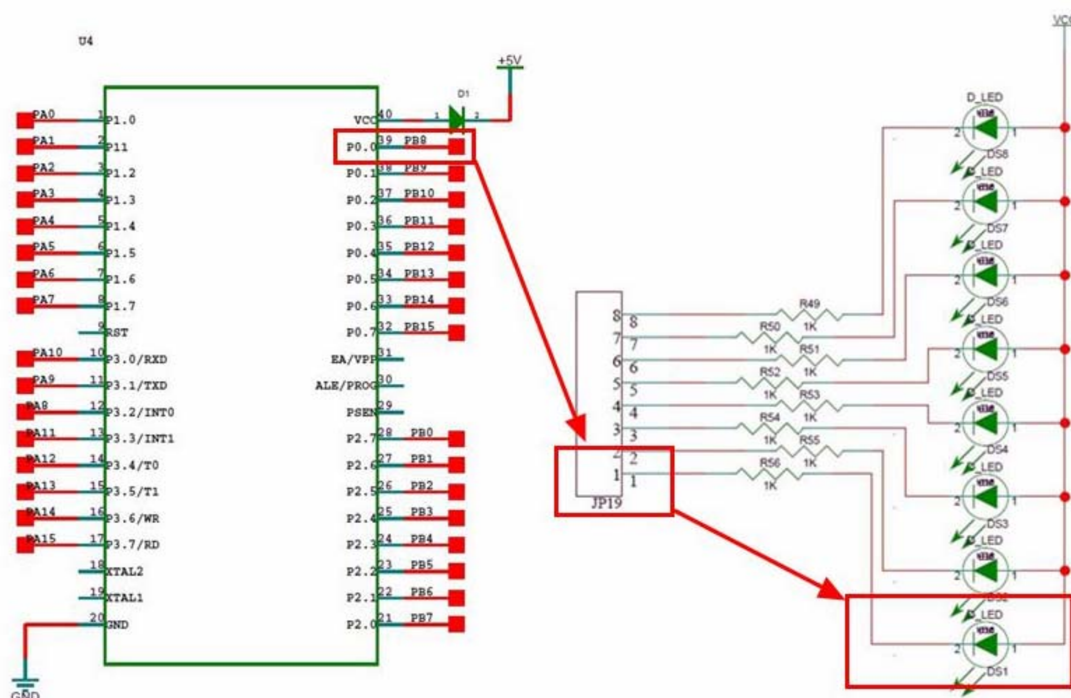


图 5-50　LED 实验相关原理图

2. 调试说明：

下载代码，并且按下【复位】键，在神舟 51+ARM 开发板可以看到用杜邦线导通的 LED 灯一亮一灭。

3. 关键代码：

```
/************ www.armjishu.com *******/
int main(void)   //main 是程序入口
{
/****  程序总共 2 部分之第 1 部分  时钟频率的配置  {开始  *******/
/**  以下是关于 RCC 时钟  详细请见《STM32F10XXX 参考手册》6.3 节 RCC 寄存器描述**/
        unsigned char sws = 0;
        RCC->CR |= 0X00010000;     //使能外部高速时钟 HSEON
         //将 RCC_CR 寄存器的值右移 17 位，等待 HSERDY 就绪，即外部时钟就绪
        while(!(RCC->CR>>17));

 /*  因为手册有要求 APB1 时钟频率不超过 36MHZ，而在 STM32 中最大为 72MHZ */
       /*  为了保证最大速度，我们这里设置成 2 分频  */
       /*  设置寄存器 CFGR 里的 8-10 位的值为 100 */
        RCC->CFGR = 0x00000400;

       /*  寄存器 CFGR 的 18-21 四个 bit 位配置成以下值,则 PLL 就会设置成对应的值：
          0000：PLL 2 倍频输出     1000：PLL 10 倍频输出
          0001：PLL 3 倍频输出     1001：PLL 11 倍频输出
          0010：PLL 4 倍频输出     1010：PLL 12 倍频输出
          0011：PLL 5 倍频输出     1011：PLL 13 倍频输出
          0100：PLL 6 倍频输出     1100：PLL 14 倍频输出
          0101：PLL 7 倍频输出     1101：PLL 15 倍频输出
          0110：PLL 8 倍频输出     1110：PLL 16 倍频输出
          0111：PLL 9 倍频输出     1111：PLL 16 倍频输出
          我们在这里，因为神舟 51+ARM 之 STM32F103C8T 开发板上的晶振是 8MHZ 的，配置成 9 倍输出就能达到 STM32 最大 72MHZ 工作频率*/
          //本例程希望设置成 32MHZ 的工作频率，我们在这里尝试一下
          RCC->CFGR |= 2<<18;
//2 右移动 18 位，即 0010 使得 PLL 获得 4 倍频输出，外部晶振是 8MHZ 乘以 4 就是 32MHZ
          RCC->CFGR |= 1<<16; //PLLSRC 设置成 1，使得 HSE 时钟作为 PLL 输入时钟
          RCC->CR |= 1<<24;     //将 PLL 使能
          while(!(RCC->CR>>25)); //监控寄存器 CR 的 PLLRDY 位，等待 PLL 时钟就绪
          RCC->CFGR |= 1<<1;   //将时钟切换寄存器配置成用 PLL 输出作为系统时钟
          while(sws != 0x2)      //等待 CFGR 寄存器的 2，3 位为 10，系统正式切换到了 PLL 输出作为时钟
```

```
            {
// 将 CFGR 寄存器右移 2 位，将 2，3 位 SWS 状态移出来，详情请见《STM32F10XXX 参考手册》54 页
                sws = RCC->CFGR>>2;
//这里的 0x3 为二进制的 11，这个 whlie 循环设计的一个算法，为了判断 sws 是不是为 10
                sws &= 0x3;
            }
/**程序总共 2 部分之第 1 部分  时钟频率的配置   结束} **/

/**  程序总共 2 部分之第 2 部分  点灯的配置  {开始**/
/*  使能 APB2 总线的时钟，对 GPIO 的端口 B 时钟使能  */
    RCC->APB2ENR |= RCC_APB2Periph_GPIOB;

    /*-- GPIO Mode Configuration 速度，输入或输出  -----------------------*/
    /*-- GPIO CRL Configuration  设置 IO 端口低 8 位的模式（输入还是输出）---*/
    /*-- GPIO CRH Configuration  设置 IO 端口高 8 位的模式（输入还是输出）---*/
    GPIOB->CRH &=   0xFFFFFFF0;
    GPIOB->CRH |=   0x00000003;
    while (1)
    {
        GPIOB->BRR = GPIO_Pin_8;
        Delay(0xFFFFFF);
        GPIOB->BSRR = GPIO_Pin_8;
        Delay(0xFFFFFF);
    }
/**  程序总共 2 部分之第 2 部分  点灯的配置   结束}   **/
}
```

这个例程主要是体现在如何设置时钟，点灯的代码和原理图都在通用输入/输出的 GPIO 章节详细说明了。

STM32 的时钟源有几种，有内部的 RC，也有外部的晶振，该选择哪种，代码里通过语句：RCC->CR |= 0X00010000；使能外部的晶振，那么 CR 这个寄存器全称叫做 STM23 的时钟控制寄存器，在《STM32F10xxx 参考手册》的 51 页可以看到 RCC_CR 的地址偏移值为 0x10，复位值为 0x0000 0000，寄存器结构及说明分别如表 5-21 所示。

表 5-21　RCC_CR 寄存器

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留						PLL RDY	PLLON	保留				CSS ON	HSE BYP	HSE RDY	HSE ON
						r	rw					rw	rw	r	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
HSICAL[7:0]								HSITRIM[4:0]					保留	HIS	HSI

														RDY	ON
r	r	r	r	r	r	r	r	rw	rw	rw	rw	rw		r	rw

再分析一下 RCC->CR |= 0X00010000 这个语句 16 进制 0x00010000 换成二进制，是第 17 位为 1，其他 31 个位都为 0，寄存器是从 0 开始，就是 16 位 HSE ON 置位，如表 5-22 所示：

表 5-22　内部 RCC_CR 寄存器第十六位描述

位 16	**HSEON:**外部高速时钟使能 由软件置“1”或清零 当进入待机和停止模式时，该位由硬件清零，关闭外部时钟。当外部 4—25MHz 时钟被用作或被选择将要作为系统时钟时，该位不能被清零。 0：HSE 振荡器关闭； 1：HSE 振荡器开启。

可以看到对 HSE ON 置位就是把外部的高速时钟使能，CPU 所需要的时钟是从外部获取；然后接下继续看另外一句代码：while(!(RCC->CR>>17))，在 while 循环里，如果为 1，while 就会一直循环，如果为 0 的话 while 就会继续执行；while 循环里还有！取反的操作，所以这句代码的意思三个是让 while 循环等到 CR 的第 17 位变成 1 才往下执行，那么看下第 17 位是什么，翻到《STM32F10xxx 参考手册》的 51 页，如表 5-23 位所示：

表 5-23　RCC_CR 寄存器第十七位描述

位 17	**HSERDY:**外部高速时钟就绪标志 有硬件置“1”来指示外部 4—25MHz 时钟已经稳定。在 HSEON 位清零后，该位需要 6 个外部 4—25MHz 时钟周期清零。 0：外部 4—25MHz 时钟没有就绪； 1：外部 4—25MHz 时钟就绪。

可以看到上图，第 17 位置 1 表示外部的高速时钟就绪好了，晶振起振稳定，CPU 从外部晶振稳定的获取时钟。

接下来，就是操作一个叫做 RCC_CFGR 寄存器，这个寄存器主要负责内部时钟的倍频（倍频就是加入输入进来是 8M 主频，经过 4 倍频后就是乘以 4 倍，等于 32M 了，所以倍频就相当于是乘）和分频（分频就相当于是除法，比如 72M 的主频，4 分频后就是 18M 了），是一个非常重要的寄存器，RCC_CFGR 的地址偏移值为 0x04，复位值为 0x0000 0000，寄存器结构及说明分别如表 5-24 所示。

表 5-24 RCC_CFGR 寄存器

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留					MCO[2:0]			保留	USB PRE	PLLMUL[3:0]				PLL XTPRE	PLL SRC
					rw	rw	rw		rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADCPRE[1:0]		PPRE2[2:0]			PPRE1[2:0]			HPRE[3:0]				SWS[1:0]		SW[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	r	r	rw	rw

这个寄存器主要负责配置 STM32 内部总线上的一些时钟参数，比如时钟切换状态（可

以知道目前哪个时钟作为系统时钟，比如有内部的时钟，也有外部的），这个寄存器还负责STM32 时钟内部总线的配置，比如 AHB 时钟总线，APB1 时钟总线，APB2 时钟总线是怎么分频（有几种分频方式，例如不分频，2 分频，4 分频，8 分频，16 分频等）；还有 ADC 模数转换的分频；USB 的分频等；可以看到这个寄存器完成了许多的功能，下面来分析我们如何操作这个寄存器的：

1）RCC->CFGR = 0x00000400;

2）RCC->CFGR |= 2<<18;

3）RCC->CFGR |= 1<<16;

4）RCC->CFGR |= 1<<1;

第 1 句代码是设置 PPRE1 寄存器为 100，从下表 5-25 可以知道，设置 HCLK2 分频，这里也可以不设置，默认 HCLK 是不分频的，我们在这里只是提醒大家，做个配置师范：

表 5-25RCC_CFGR 寄存器 PPRE1 位描述

位 10：8	**PPRE1:**低速 APB 预分频（APB1） 有软件置“1”或清“0”来控制低速 APB1 时钟（PCLK1）的预分频系数。 注意：软件必须保证 APB1 时钟频率不超过 36MHz。 0xx：HCLK 不分频 100：HCLK2 分频 101：HCLK4 分频 110：HCLK8 分频 111：HCLK16 分频

第 2 句代码是将 2 左移 18 位，即将二进制的 10 左边移动 18 位，是 PLL 4 倍频输出，如表 5-26 所示：

表 5-26　RCC_CFGR 寄存器 PLLMUL 位描述

位 21：18	**PLLMUL:**PLL 倍频系数 由软件设置来确定 PLL 倍频系数。只有在 PLL 关闭的情况下才可被写入。 注意：PLL 的输出频率不能超过 72MHz 0000：PLL2 倍频输出　　1000：PLL10 倍频输出 0001：PLL3 倍频输出　　1001：PLL11 倍频输出 0010：PLL4 倍频输出　　1010：PLL12 倍频输出 0011：PLL5 倍频输出　　1011：PLL13 倍频输出 0100：PLL6 倍频输出　　1100：PLL14 倍频输出 0101：PLL7 倍频输出　　1101：PLL15 倍频输出 0110：PLL8 倍频输出　　1110：PLL16 倍频输出 0111：PLL9 倍频输出　　1111：PLL16 倍频输出

第 3 句代码是左移 16 位后置 1，表示 HSE 时钟作为 PLL 输入时钟，HSE 就是外部晶振，如表 5-27 所示：

表 5-27 RCC_CFGR 寄存器 PLLSRC 位描述

位 16	**PLLSRC**：PLL 输入时钟源 由软件置“1”或清“0”来选择 PLL 输时钟源。该位只有在 PLL 关闭时才可以被写入。

	0：HSI 时钟 2 分频后作为 PLL 输入时钟 1：HSE 时钟作为 PLL 输入时钟。

第 4 句代码表示将时钟切换到外部时钟，都设置好之后，通知 STM32 的芯片可以将系统的时钟切换给外部晶振提供时钟了，如表 5-28 所示：

表 5-28 RCC_CFGR 寄存器 SW 位描述

位 1：0	**SW：**系统时钟切换 由软件置“1”或清“0”来选择系统时钟源。 在从停止或待机模式中返回时或直接或间接作为系统时钟的 HSE 出现故障时，由硬件强制选择 HIS 作为系统时钟（如果时钟安全系统已经启动） 00：HIS 作为系统时钟； 01：HSE 作为系统时钟； 10：PLL 输出作为系统时钟； 11：不可用。

可以从下图 5-51 看到，RCC_CFGR 寄存器里的 PLLSRC 负责切换是内部 HIS 提供时钟，还是外部 HSE 晶振提供时钟，上面程序代码选择的是外部的 HSE 提供时钟；紧接着，RCC_CFGR 寄存器里的 PLLMUL，程序代码里配置的是 4 倍频，如果这里外部晶振是 8MHz 的，4 倍频后就是 32MHz 的频率（STM32 最高可以达到 72MHz）；RCC_CFGR 寄存器里的 SW 负责 HIS、PLLCLK、HSE 三个时钟来源取其中一个，程序里选择的是 PLLCLK，这个来自 HSE 外部晶振然后 4 倍频的时钟做为整个 STM32 芯片的主时钟。

图 5-51 时钟树配置路径

上图上面已经经过详细分析了，接下来就是一个 while(sws != 0x2)循环

```
while(sws != 0x2)
{
    sws = RCC->CFGR>>2;
    sws &= 0x3;
}
```

上面的代码是将 RCC_CFGR 寄存器右移 2 位后，在与上 0x3 相当于与上二进制的 11，最低两位是 1，其他高位都是 0；这样的意思就是把 SWS 这个系统时钟切换状态的值单独截取出来，然后用 while(sws != 0x2)语句一直等待 SWS 的值为 0x2，0x2 化成二进制是 10，就是 PLL 输出作为系统时钟，下表 5-29 所示可以知道：

表 5-29 RCC_CFGR 寄存器 SWS 位描述

位 3：2	**SWS：** 系统时钟切换状态 由硬件置“1”或清“0”来指示哪一个时钟源被作为系统时钟。 00：HIS 作为系统时钟； 01：HSE 作为系统时钟； 10：PLL 输出作为系统时钟； 11：不可用。

那么这里就奇怪了，为什么不是 HSE 的晶振作为系统时钟呢？而是采用 PLL，因为我们从图 5-52 可以看到，我们在代码里设置的就是走左边这张图的配置路线，因为把外部的晶振进行倍频了的，也有右边这种选择，只是这个例程我们在代码里没有这么设计，以后大家可以这样设计：

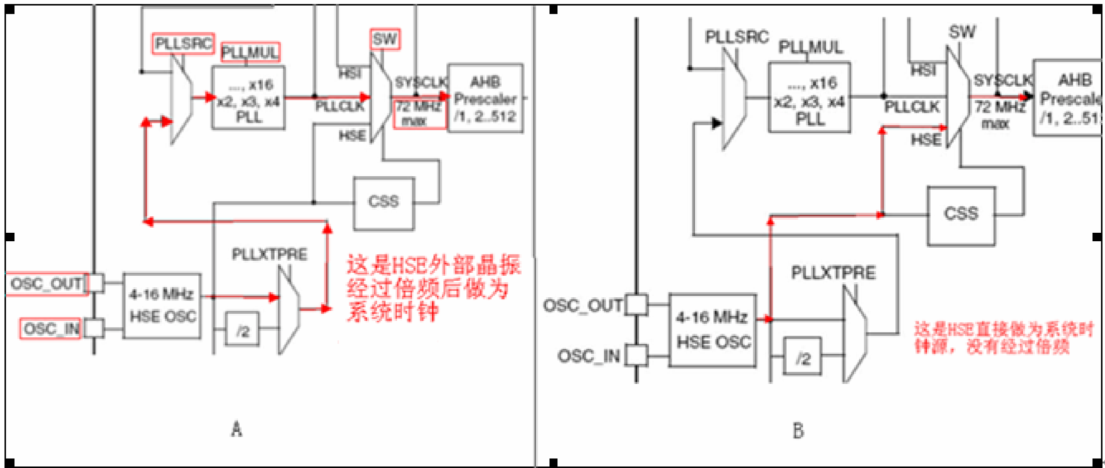


图 5-52 时钟树配置

最后代码配置好了，就开始点灯程序，具体细节请见代码，因为上一章节已经有详细分析。

5.4.5 例程02 STM32芯片40MHZ频率下跑点灯程序

1. 示例简介

   让点灯程序在时钟主频 40MHz 下面运行，其他不变，同上个例程一样，唯一的改变就是把主频从 32MHz 改成 40MHz 了。

2. 调试说明：

下载代码，并且按下【复位】键，在神舟 51+ARM 开发板上可以看到该 LED 灯一亮一灭。

3. 关键代码：

查看 RCC_CFGR 寄存器的第 18~21 位的 PLLMUL，如表 5-30 所示，当外部晶振是 8MHz 时，将 RCC->CFGR |= 3<<18;将 16 进制的 3 化成二进制是 0011，把 PLL 设置成 5 倍频输出，8MHz 乘以 5 倍频就是 40MHz。

表 5-30 RCC_CFGR 寄存器 PLLMUL 位描述

位 21：18	**PLLMUL:**PLL 倍频系数 由软件设置来确定 PLL 倍频系数。只有在 PLL 关闭的情况下才可被写入。 注意：PLL 的输出频率不能超过 72MHz 0000：PLL2 倍频输出　1000：PLL10 倍频输出 0001：PLL3 倍频输出　1001：PLL11 倍频输出 0010：PLL4 倍频输出　1010：PLL12 倍频输出 0011：PLL5 倍频输出　1011：PLL13 倍频输出 0100：PLL6 倍频输出　1100：PLL14 倍频输出 0101：PLL7 倍频输出　1101：PLL15 倍频输出 0110：PLL8 倍频输出　1110：PLL16 倍频输出 0111：PLL9 倍频输出　1111：PLL16 倍频输出

关键代码的改变与上面一模一样，唯一的改变就是将 RCC->CFGR |= 2<<18;变成了 RCC->CFGR |= 3<<18;这句不同

5.4.6 例程03 STM32芯片72MHZ频率下跑点灯程序

1.示例简介

让点灯程序在时钟主频 72MHz 下面运行，其他不变，同上个例程一样，唯一的改变就是把主频从 40MHz 改成 72MHz 了。

2.调试说明：

下载代码，并且按下【复位】键，在神舟 51+ARM 开发板上可以看到该 LED 灯一亮一灭。

3.关键代码：

查看 RCC_CFGR 寄存器的第 18~21 位的 PLLMUL 如表 5-31，当外部晶振是 8MHz 时，将 RCC->CFGR |= 7<<18;;将 16 进制的 7 化成二进制是 0111，把 PLL 设置成 9 倍频输出，8MHz 乘以 9 倍频就是 72MHz。

表 5-31 RCC_CFGR 寄存器 PLLMUL 位描述

位 21：18	**PLLMUL:**PLL 倍频系数 由软件设置来确定 PLL 倍频系数。只有在 PLL 关闭的情况下才可被写入。 注意：PLL 的输出频率不能超过 72MHz

	0000：PLL2 倍频输出	1000：PLL10 倍频输出
	0001：PLL3 倍频输出	1001：PLL11 倍频输出
	0010：PLL4 倍频输出	1010：PLL12 倍频输出
	0011：PLL5 倍频输出	1011：PLL13 倍频输出
	0100：PLL6 倍频输出	1100：PLL14 倍频输出
	0101：PLL7 倍频输出	1101：PLL15 倍频输出
	0110：PLL8 倍频输出	1110：PLL16 倍频输出
	0111：PLL9 倍频输出	1111：PLL16 倍频输出

关键代码的改变与上面一模一样，唯一的改变就是将 RCC->CFGR |= 3<<18;变成了 RCC->CFGR |= 7<<18;这句不同。

5.5 独立按键

5.5.1 按键的分类

目前，按键有多种形式。有机械接触式，电容式，轻触式等。

1.按制作工艺分：

硬板按键：带弹簧的按键焊接在印刷电路板上

软板键盘：以导电橡胶作为接触材料放在以聚脂薄膜作为基底的印刷电路上所形成的按键。

2.按工艺原理分：

可以将键盘分为编码键盘和非编码键盘，编码键盘的键盘电路内包含有硬件编码器，当按下某一个键后，键盘电路能直接提供与 该键相对应的编码信息，例如 ASCII 码。 非编码键盘的键盘电路中只有较简单的硬件，采用软件来识别按下键的位置，并提供与按下 键相对应的中间代码送主机，然后由软件将中间代码转换成相应的字符编码，例如 ASCII 码；非编码键盘主要靠软件编程来识别的，在单片机组成的各种系统中，用的较多的是非编码键盘。非编码键盘又分为独立键盘和行列式（又称矩阵式）键盘。

5.5.2 按键属性

键盘实际上就是一组按键，在单片机外围电路中，通常用到的按键都是机械弹性开关，当开关闭合时，线路导通，开关断开时，线路断开，下图 5-53 所示是几种单片机系统常见的按键：



图 5-53 单片机系统常见按键

弹性小按键被按下时闭合，松手后自动断开；自锁式按键按下时闭合且会自动锁住，只有再次按下时才弹起断开。

单片机的外围输入控制用小弹性按键较好，单片机检测按键的原理是：单片机的 I/O 口既可作为输出也可作为输入使用，当检测按键时用的是它的输入功能，我们把按键的一端接地，另一端与单片机的某个 I/O 口相连，开始时先给该 I/O 口赋一高电平，然后让单片机不断地检测该 I/O 口是否变为低电平，当按键闭合时，即相当于该 I/O 口通过按键与地相连，变成低电平，程序一旦检测到 I / O 口变为低电平则说明按键被按下，然后执行相应的指令，下图是按键动作的波形变化图：

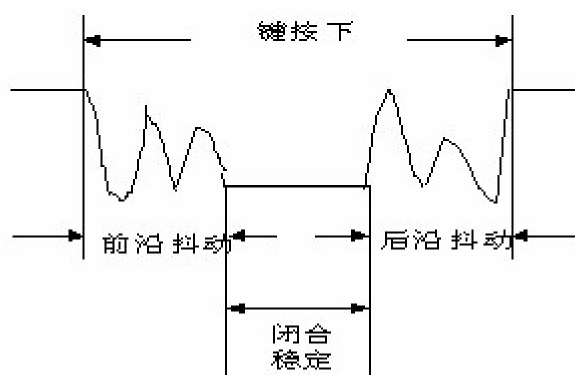


图 5-54 按键动作波形的变化

从上图 5-54 可看出，理想波形与实际波形之间是有区别的，为什么呢？因为实际波形在按下和释放的瞬间会有抖动现象出现，这是因为通常的按键所用开关为机械弹性开关，当机械触点断开、闭合时，由于机械触点的弹性作用，一个按键开关在闭合时不会马上稳定地接通，在断开时也不会一下子断开。因而在闭合及断开的瞬间均伴随有一连串的抖动，抖动时间的长短和按键的机械特性有关，一般为 5～l0ms。为了不产生这种现象而作的措施就是按键消抖。通常我们手动按下键然后立即释放，这个动作中稳定闭合的时间超过 20ms。因此单片机在检测键盘是否按下时都要加上去抖动操作。如图 5-55 所示

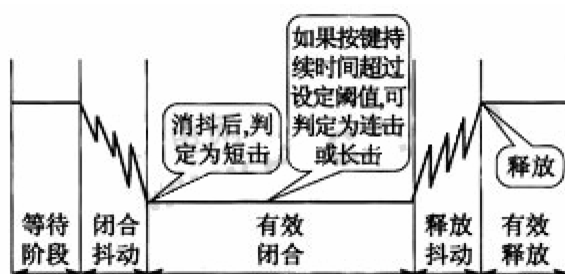


图 5-55　按键的抖动

消抖是为了避免在按键按下或是抬起时电平剧烈抖动带来的影响。按键的消抖，可用硬件或软件两种方法，硬件的去抖主要是用专用的去抖动电路，也有专用的去抖动芯片；另外一种方式就是用软件延时的方法就能很容易解决抖动问题，而没有必要再添加多余的硬件电路。所以软件消抖适合按键比较多的情况，而硬件消抖适合按键比较少的情况。

如果按键较多，就用软件方法去抖，即检测出键闭合后执行一个延时程序，5ms～10ms 的延时，让前沿抖动消失后再一次检测键的状态，如果仍保持闭合状态电平，则确认为真正有键按下。当检测到按键释放后，也要给 5ms～10ms 的延时，待后沿抖动消失后才能转入该键的处理程序，这样就靠软件模拟整个按键的过程，控制只取最稳定的那个按键状态。

实现方法：一般来说，软件消抖的方法是不断检测按键值，直到按键值稳定。假设检测到按键按下之后，为了避免检测到很多的抖动，可以先延时 5ms～10ms，再次检测，如果按键还被检测按下，那么就认为有一次按键输入（因为如果不避开抖动的话，会有很多次按键输入信号出现，通过去抖，模拟人的按下的过程和时间，按下和松开按键实际也占用了 20ms 以上的时间，记录正确的按键次数。

5.5.3 STM32的位带操作

1. 什么是位带操作

还记得 51 单片机吗？ 单片机 51 中也有位的操作，以一位（BIT）为数据对象的操作；例如 51 单片机可以简单的将 P1 端口的第 2 位独立操作，P1.2=0 或者 P1.2=1，就是这样把 P1 口的第三个脚（bit2）置 0（输出低电平）或者置 1（输出高电平）。

而现在 STM32 的位段、位带别名区这些就是为了实现这样的功能，可以在 SRAM、I/O 外设空间实现对这些区域的某一位的单独直接操作。

2. 为什么要用位带操作？

那么 51 单片机中间不是有位的操作吗，而 STM32 为什么要提出位带的操作呢？首先，这里不得不提一个事情就是 STM32 的内部区域访问只能是 32 位的字，不能是字节或者半字，这部分 STM32 在神舟开发板手册的 GPIO 章节中提到过；而 51 单片机里一个 bit（一个字节等于 8 个 bit，一个字是 32 个 bit）。这个是 STM32 的特点决定的，所以 STM32 使用一种新型的方式来解决这个问题，设计一个办法来解决用一次访问 32bit 的这样的操作达到 51 单片机那种只访问一个 bit 的效果。

3. 如何设计和实现位带操作的？

从编程者这个角度来说，我们操作的对象是一个一个的 bit 位，而对于 STM32 来说，它内内部只能是 32 位 bit 每次的访问。如果要实现这个技术，必须要做一个映射，也就是从 1 个 bit 映射到 32 个 bit，就是用 STM32 内部的一次访问（32 个 bit）来代表编程者认为的 1 个 bit。

那 STM32 内部是如何解决的呢？它是在支持位带操作的地方，取个别名区空间，而这个别名区空间可以让一次 32 位来进行访问，对这个别名进行操作就相当于对 SRAM 或者 I/O 存储空间中的位（1 个寄存器里的位就是 1 个 bit，1 个 bit 最后对应别名区空间的 32 个位，因为 STM32 芯片内部只能是 32 位去访问）进行操作。如图 5-56 所示：

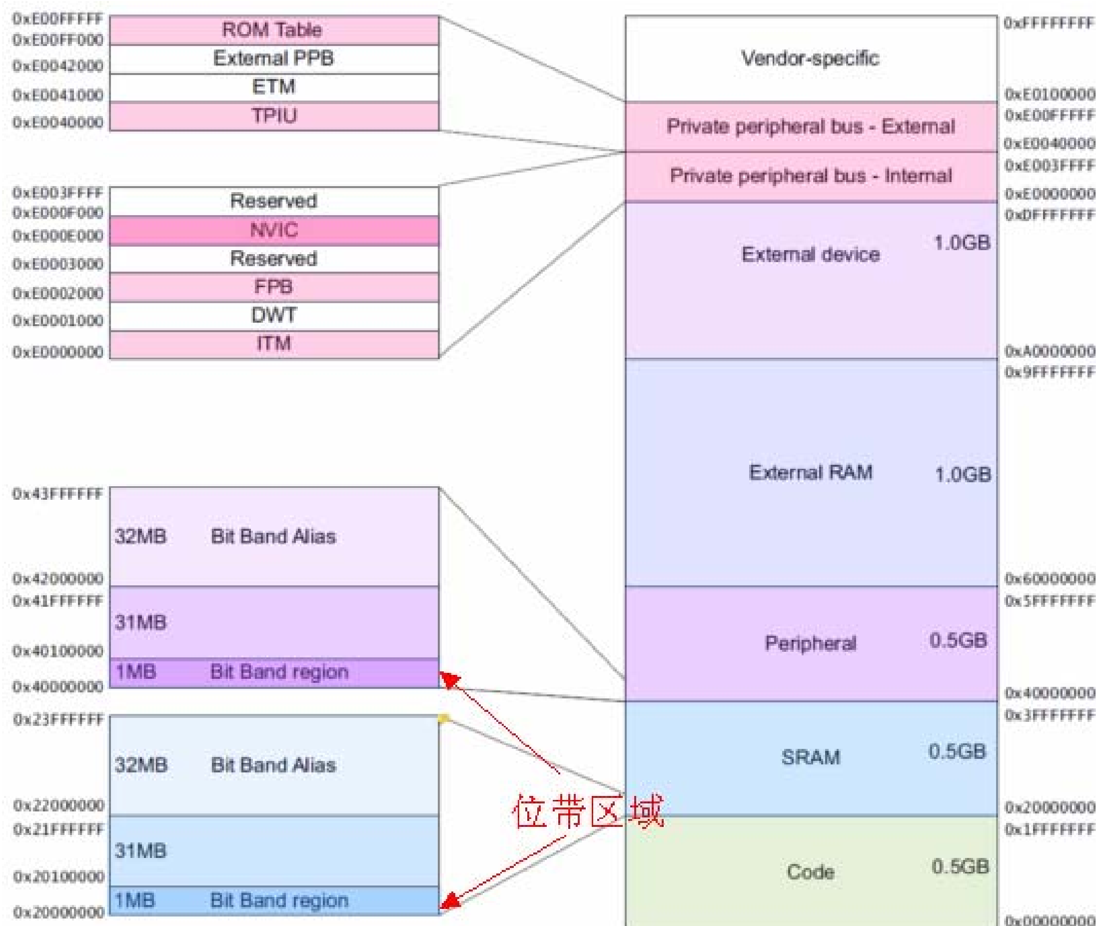


图 5-56　位带操作，别名区空间

这样呢，1MB SRAM 就可以 32M 个对应别名区空间，就是 1 位膨胀到 32 位（1bit 变为 1 个字）；我们对这个别名区空间开始的某一字操作，置 0 或置 1，就等于它映射的 SRAM 或 I/O 相应的某地址的某一位的操作。

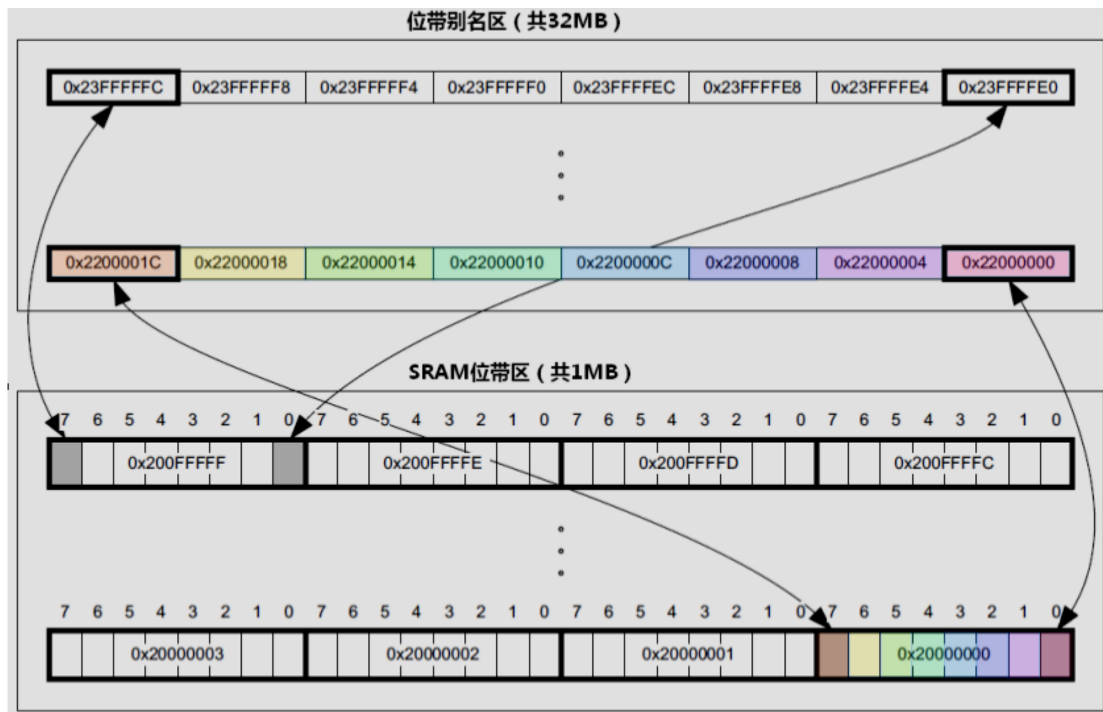
4.　STM32 中位带操作的具体部署情况是

支持位带操作的两个内存区的范围如下表 5-32：

表 5-32　位带操作 内存区范围

序号	支持位带操作的两个内存区的范围	对应的别名区空间范围
1	SRAM 区中的最低 1MB： 0x2000_0000-0x200F_FFFF	SRAM 所对应的别名区 32MB 空间： 0x2200_0000-0x23FF_FFFF
2	片上外设区中的最低 1MB： 0x4000_0000-0x400F_FFFF	片上外社区所对应的别名区 32MB 空间： 0x4200_0000-0x43FF_FFFF

下面图 5-57 是内部空间映射图：



5-57 内部空间映射图

例如：SRAM 区中的最低 1MB 空间中的 0x2000_0000 的 8 个 bit，分别对应如下表 5-33：

表 5-33 SRAM8 个 bit 对应

地址	对应的 bit 位	别名空间	Bit 对应的别名空间
0x2000_00000	Bit1	0x2200_0000	跨度一个字 ＝4 个字节 ＝32 个 bit
	Bit2	0x2200_0004	跨度一个字 ＝4 个字节 ＝32 个 bit
	Bit3	0x2200_0008	跨度一个字 ＝4 个字节 ＝32 个 bit
	Bit4	0x2200_000C	跨度一个字 ＝4 个字节 ＝32 个 bit
	Bit5	0x2200_0010	跨度一个字 ＝4 个字节 ＝32 个 bit
	Bit6	0x2200_0014	跨度一个字 ＝4 个字节 ＝32 个 bit
	Bit7	0x2200_0018	跨度一个字 ＝4 个字节 ＝32 个 bit
	Bit8	0x2200_001C	跨度一个字 ＝4 个字节 ＝32 个 bit

可以看到上表和上图，0x2000_0000 中的一个 bit 位对应了别名区的一个 32 位的字，也就是说 STM32 芯片的内部寄存器的任意一个位，都其实对应的是别名区的 32 个位。

5. 如何用代码与位带操作挂钩

在 STM32 中，一个寄存器是 32 位的，32 个 bit 中的任意其中一个 bit 所对应的别名空间到底该如何访问呢？首先分为两种情况，一种是在 SRAM，一种是在 FLASH 中，两个别名空间的起地位置是不同的，SRAM 是从 0x2200_0000 开始，而 FLASH 是从 0x4200_0000 开始。

假如在 SRAM 中的一个寄存器的地址是 A，访问寄存器 A 中的第 n 个 bit 位。那么该如何计算呢？我们知道 SRAM 中别名区的起始地址是 0x2200_0000 对应 SRAM 中实际寄存器地址 0x2000_0000，SRAM 中每 1 个 bit，都会对应别名区中的 32 个 bit，那么实际地址的公式应该如下：

0x2200_0000 + （SRAM 实际寄存器地址偏移 0x2000_0000 的 bit 数）* 4

因为 0x2200_0000 这个地址每增加 1，实际上就是增加 8 个 bit（一个地址对应

一个字节），实际寄存器中的 1 个 bit 对应 32 个 bit，所以就乘以 4，地址本身增加 1 是 8bit，8bit 乘以 4 倍刚好是 32bit。

那么接下来" SRAM 实际寄存器地址偏移 0x2000_0000 的 bit 数"该如何计算呢？对，用寄存器的地址减去这个基地址，然后在乘以 8（因为一个地址对应 8 个 bit），所以就可以得到以下的公式：

（A－0x20000000)*8

以上这个公式可以知道实际寄存器离基地址有多少个 bit 的距离，访问该寄存器的第 n 个 bit 位还必须加上一个 n,就变成以下的公式：

（A－0x20000000)*8+n

好了，最后整理整个换算公式如下，FLASH 与 SRAM 的原理都是想通的：

SRAM :0x22000000 +((A－0x20000000)*8+n)*4

FLASH :0x42000000 +((A－0x40000000)*8+n)*4

6. 举例说明：

比如我要访问如下寄存器 GPIOB_BSRR 中的第 14bit 位 BS13，注意因为寄存器内部是从 0 开始计数到 31 截止，所以第 14bit 相当于是第 13。GPIOB_BSRR 寄存器的地址偏移值为 0x10，复位值为 0x0000 0000，寄存器结构及说明分别如表 5-34 和表 5-35 所示：

介绍如下表 5-34 和表 3-35 所示

表 5-34　GPIOB_BSRR 寄存器

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR 14	BR 13	BR 12	BR 11	BR 10	BR 9	BR 8	BR 7	BR 6	BR 5	BR 4	BR 3	BR 2	BR 1	BR 0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS 14	BS 13	BS 12	BS 11	BS 10	BS 9	BS 8	BS 7	BS 6	BS 5	BS 4	BS 3	BS 2	BS 1	BS 0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

表 5-35　GPIOB_BSRR 寄存器各个位功能说明

位	说明
位 31:16	**BRy**：清除端口 x 的位 y（y=0...15） 这些位只能写入并只能以字（16 位）的形式操作。 0：对对应的 ODRy 位不产生影响 1：清除对应的 ODRy 位为 0 注：如果同时设置了 BSy 和 BRy 的对应位，BSy 位起作用。
位 15:0	**BSy**：设置端口 x 的位 y（y=0...15） 这些位只能写入并只能以字（16 位）的形式操作。 0：对对应的 ODRy 位不产生影响 1：清除对应的 ODRy 位为 1

可以从下表 5-36 看到，GPIO 端口 B 的起始地址是 x04001_0C00，GPIOB_BSRR 寄存器的偏移地址是 0x10，访问的第 14bit 位的 BS13。

表 5-36GPIO 端口地址

0x4001 1C00		
0x4001 1800	Port E	1 Kbit
0x4001 1400	Port D	1 Kbit
0x4001 1000	Port C	1 Kbit
0x4001 0C00	Port B	1 Kbit
0x4001 0800	Port A	1 Kbit
0x4001 0400	EXTI	1 Kbit
0x4001 0000	AFIO	1 Kbit

那么通过公式：

FLASH :0x42000000 +((A ‐ 0x40000000)*8+n)*4

换算 0x4200_0000 + ((0x40010c00-0x40000000)*8 + 12)*4 = 实际地址

在这里我们就不具体计算了,SRAM 访问也是同理

7. 如何将理念转化成代码：

由上面几节得出，SRAM 和 FLASH 中别名区的寻址公式如下：

SRAM :0x22000000 +((A ‐ 0x20000000)*8+n)*4

FLASH :0x42000000 +((A ‐ 0x40000000)*8+n)*4

可以看到 0x2200_0000 和 0x4200_0000 都共同有个 x200_0000 这个数值;乘以 8 相当于再加上外面的那个乘以 4，总共是乘以 32，而 n 是乘以 4；乘以 32 相当于是数值向左移 5 位，乘以 4 相当于向左边移 2 位。

尝试将如下公式化成

```
#define BITBAND(addr，bitnum)
    ((addr & 0xF0000000)+0x2000000+((addr &0xFFFFF)<<5)+(bitnum<<2))
```

这样就可以用 BITBAND(addr，bitnum)来表示寄存器里的任何一个 bit 位，大概原理讲到这里，具体使用方法通过例程来体现。

5.5.4 例程01 STM32芯片按键点灯（无防抖）

1. 示例简介

先用一根杜邦线将 PA0 和 JP18 中的 8 脚连起来。通过连上 PA0 管脚的按键，学会如何从 PA0 管脚读入一个输入，如何配置 GPIO 口成输入状态；当 PA0 获取到输入的数据，STM32 对 LED 灯的状态进行取反。按键 SW3 是一端连接了 GND，一端连接了 PA0 管脚；当按键按下时，PA0 管脚的电平值被拉低，相关电路图如下图 5-60 所示：

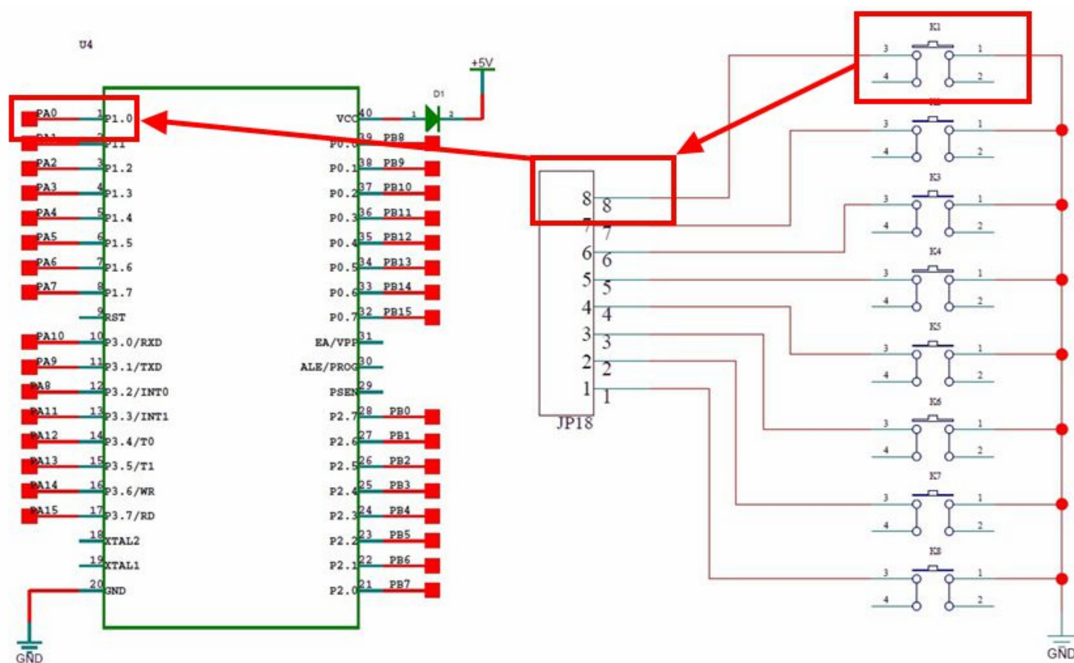


图 5-58 LED 实验相关原理图

2．调试说明：

按下 PA0 管脚所连的按键（按钮 1），每按一次，LED 灯会由亮变灭，或者又灭变亮，因为没有防抖代码（下个例程会增加），会发现，有时候按下去，灯会亮灭好几次。

3．关键代码：

```
int main(void)   //main 是程序入口
{
    unsigned int key_up =1;
    RCC_init();          //初始化配置时钟频率为 72MHZ
    LED_init();          //LED 初始化配置
    Key_init();          //初始化控制按键的 PA0 端口

    while (1)
    {
            if(key_up)
                LEDON;                  //  开灯
            else
                LEDOFF;                 //  关灯

            If ( KEY0 == 0)
                key_up = !key_up;
    }
}
```

STM32 芯片的 GPIO 管脚要采集到按键按下，那么它需要被配置成输入模式，之前有讨论 GPIO 管脚的几种模式，我们接下来还是复习一下，我们看看端口低配置寄存器 CRL 的地址偏移值为 0x00，复位值为 0x4444 4444，寄存器结构及说明分别如表 5-37 和表 5-38

所示：

表 5-37 端口低配置寄存器 CRL 描述

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF7[1:0]		MODE7[1:0]		CNF6[1:0]		MODE6[1:0]		CNF5[1:0]		MODE5[1:0]		CNF4[1:0]		MODE4[1:0]	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNF3[1:0]		MODE3[1:0]		CNF2[1:0]		MODE2[1:0]		CNF1[1:0]		MODE1[1:0]		CNF0[1:0]		MODE0[1:0]	

表 5-38 端口低配置寄存器 CRL（功能说明）

位31:30 27:26 23:22 19:18 15:14 11:10 7:6 3:2	**CNFy[1:0]**：端口x配置位(y = 0…7) 软件通过这些位配置相应的I/O端口，请参考表15端口位配置表。 在输入模式(MODE[1:0]=00)： 00：模拟输入模式 01：浮空输入模式(复位后的状态) 10：上拉/下拉输入模式 11：保留 在输出模式(MODE[1:0]>00)： 00：通用推挽输出模式 01：通用开漏输出模式 10：复用功能推挽输出模式 11：复用功能开漏输出模式
位29:28 25:24 21:20 17:16 13:12 9:8, 5:4 1:0	**MODEy[1:0]**：端口x的模式位(y = 0…7) 软件通过这些位配置相应的I/O端口，请参考表15端口位配置表。 00：输入模式(复位后的状态) 01：输出模式，最大速度10MHz 10：输出模式，最大速度2MHz 11：输出模式，最大速度 50MHz

该寄存器的复位值为 0X4444 4444（4 化成二进制为 0100），从上图可以看到，复位值其实就是配置端口为浮空输入模式。从上图还可以得出：STM32 的 CRL 控制着每个 IO 端口（A~G）的低 8 位的模式。每个 IO 端口的位占用 CRL 的 4 个位，高两位为 CNF，低两位为 MODE。这里我们可以记住几个常用的配置，比如 0X0 表示模拟输入模式（ADC 用）、0X3 表示推挽输出模式（做输出口用，50M 速率）、0X8 表示上/下拉输入模式（做输入口用）、0XB 表示复用输出（使用 IO 口的第二功能）。

STM32 的 IO 口位配置表如下表 5-39 所示：

表 5-39 IO 口位配置表

配置模式	CNF1	CNF0	MODE1	MODE0	PxODR 寄

					存器
通用输出	推挽式（Push-Pull）	0	0	01	0 或 1
	开漏（Open-Drain）		1	10	0 或 1
复用功能输出	推挽式（Push-Pull）	1	0	11	不使用
	开漏（Open-Drain）		1	见表 3.1.2	不使用
输入	模拟输入	0	0	00	不使用
	浮空输入		1		不使用
	下拉输入	1	0		0
	上拉输入				1

可以看到 CNFX 是上面 CRL 寄存器里的配置位，这里配置位的不同，就会产生不同的 GPIO 管脚模式。

STM32 输出模式配置如下表 5-40 所示：

表 5-40 STM32 输出模式配置

MODE[1:0]	**意义**
00	保留
01	最大输出速度为 10MHz
10	最大输出速度为 2MHz
11	最大输出速度为 50MHz

这里 CRL 寄存器的 MODE 配置位的选项，不同的配置就会产生不同的速率。

CRH 的作用和 CRL 完全一样，只是 CRL 控制的是低 8 位输出口，而 CRH 控制的是高 8 位输出口，大家可以自己看 STM32 手册，我们在这里 CRH 就不做详细介绍了。

首先看下代码：RCC_init();这里实现的是初始化配置时钟频率为 72MHZ，之前的时钟章节已经将过了，具体细节可以看上一章的内容，有详细的介绍。

下面看下连接按键的 GPIO 管脚的具体设置，这个函数里有 3 句代码：

```
void Key_init()
{
    RCC->APB2ENR |= RCC_APB2Periph_GPIOA;        //使能 PORTA 时钟
    GPIOA->CRL &= 0XFFFFFFF0;
    GPIOA->CRL |= 0X00000008;   //PA0 设置成输入,PA0 在按键原理图默认被上拉的
}
```

RCC->APB2ENR |= RCC_APB2Periph_GPIOA 首先使能 PORTA 的时钟，接下来开始使用 PA0 管脚，所以要先初始化 PA 端口的时钟，才可以使用，PA0 默认是被上拉，高电平的，GPIOA->CRL &= 0XFFFFFFF0;这句代码是将 PA0 的 GPIO_CRL 寄存器的前 4 位清 0，然后 GPIOA->CRL |= 0X00000008 这句话是将 GPIO_CRL 寄存器的前 4 位赋值 0x8，化成二进制就是 1000，通过寄存器查询如下表 5-41 和表 5-42：

表 5-41 GPIOA_CRL 寄存器描述

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF7[1:0]		MODE7[1:0]		CNF6[1:0]		MODE6[1:0]		CNF5[1:0]		MODE5[1:0]		CNF4[1:0]		MODE4[1:0]	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNF3[1:0]		MODE3[1:0]		CNF2[1:0]		MODE2[1:0]		CNF1[1:0]		MODE1[1:0]		CNF0[1:0]		MODE0[1:0]	

表 5-42 GPIOA_CRL 寄存器（功能说明）

位	说明
位31:30 27:26 23:22 19:18 15:14 11:10 7:6 3:2	**CNFy[1:0]**：端口x配置位(y = 0…7) 软件通过这些位配置相应的I/O端口，请参考表15端口位配置表。 在输入模式(MODE[1:0]=00)： 00：模拟输入模式 01：浮空输入模式(复位后的状态) 10：上拉/下拉输入模式 11：保留 在输出模式(MODE[1:0]>00)： 00：通用推挽输出模式 01：通用开漏输出模式 10：复用功能推挽输出模式 11：复用功能开漏输出模式
位29:28 25:24 21:20 17:16 13:12 9:8, 5:4 1:0	**MODEy[1:0]**：端口x的模式位(y = 0…7) 软件通过这些位配置相应的I/O端口，请参考表15端口位配置表。 00：输入模式(复位后的状态) 01：输出模式，最大速度10MHz 10：输出模式，最大速度2MHz 11：输出模式，最大速度 50MHz

可以知道将 GPIOA_CRL 寄存器的 PA0 管脚配置成 1000，即上拉/下拉输入模式，配置好 PA0 管脚之后，如何才能知道按键按下呢？这时就需要时刻查看和监听 PA0 管脚是否有电平的变化，那就要知道 PA0 的管脚的电平值，而且还要不停的去检测它是否有变化，因为我们希望当按键按下的时候，我们能以最快的反映速度获取到这一动作。

在代码里，我们使用 if (KEY0==0)来判断按键是否按下了，KEY0 就是 PA0，如果按键按下，从原理图可以知道 PA0 被拉到 GND 变成低电平，这样 PA0 就等于 0 即 KEY0 等于 0。

那么 KEY0 如何反映到 PA0 的值的呢？接下来继续分析代码：

```
（1）#define BITBAND(addr, bitnum)
     ((addr & 0xF0000000)+0x2000000+((addr &0xFFFFF)<<5)+(bitnum<<2))
（2）#define   MEM_ADDR(addr)    *((volatile unsigned long  *)(addr))
（3）#define   BIT_ADDR(addr, bitnum)        MEM_ADDR(BITBAND(addr, bitnum))
（4）#define   PAin(n)      BIT_ADDR(GPIOA_IDR_Addr,n)
（5）#define   KEY0     PAin(0)
```

代码（1)：上面有讲解，这个 BITBAND(addr,bitnum)最后得出的是某寄存器里的某个 bit 位所映射的别名区的地址。

代码（2)：unsigned long *表示强制把这个地址变成一个 32 位的长的地址的指针，或者说是一个指针，指向 32 位的一个地址；然后*((volatile unsigned long *)(addr))这个表示这个地址，长达 32 位 bit 的值，通过 MEM_ADDR(addr) 把从 addr 地址开始的连续 32 个 bit 里的值给取出来。

代码（3)：BIT_ADDR(addr, bitnum)表示某寄存器的地址，对应的某 bit 位所对应的别名区

的空间的值，也就是说取出某寄存器的某 bit 位的值

代码（4)：把 PA 的某个管脚，与 BIT_ADDR 进行绑定关联

代码(5)：把 KEY0 设置成 PA0，使得 KEY0 能取到 GPIOA_IDR 寄存器里第 0 位也就是 PA0 的值

这样，这个例程的关键是是否取到了 PA0 的值，取到了值之后，再进行相关的操作，这个是大家可以自己定义的，在这个例程中，按一下按键，我们就将 LED 灯取反，原来是亮的就变成灭的，原来是灭的就变成亮的。

5.5.5 例程02 STM32芯片按键点灯-增加了防抖的代码

1. 示例简介：

   其他都与上个例程相同，唯一不同的就是增加了防抖代码，在这里是用软件防抖

2. 调试说明：

按下 PA0 管脚所连的按键（按钮 1），每按一次，LED 灯会由亮变灭，或者又灭变亮，因为增加了防抖代码，基本上可以做到按一次，就采集到一次数据，灯由亮变灭或者又灭变亮，非常的稳定。

3. 关键代码：

原理图还是同上个例程一样，如下图 5-59 所示：

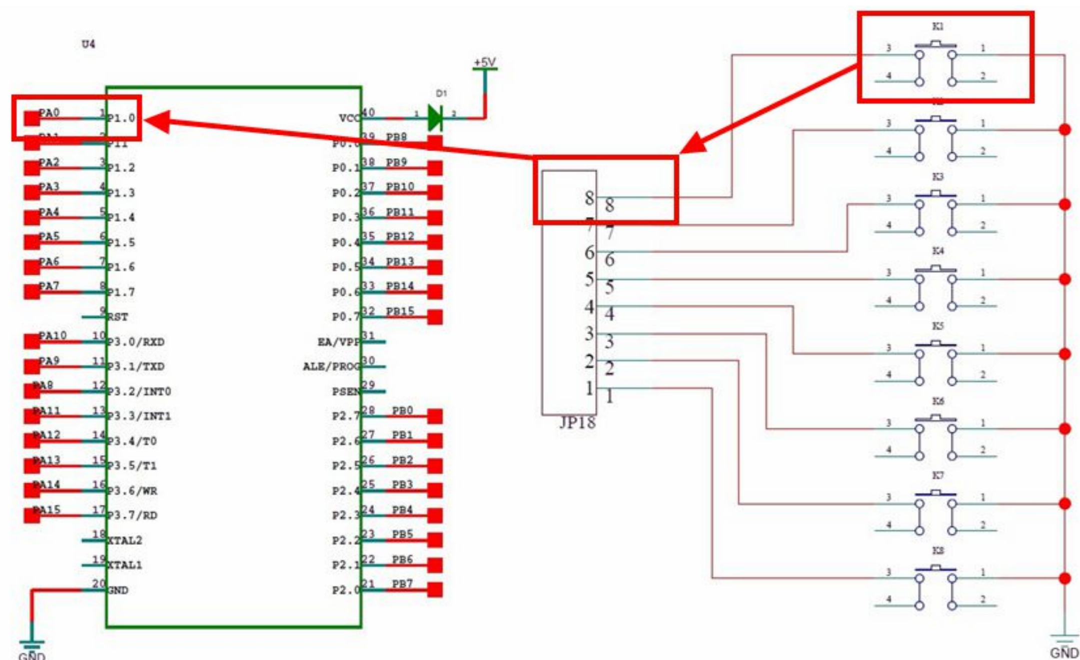


图 5-59 LED 实验相关原理图

```
int main(void)    //main是程序入口
{
    unsigned int key_up =1;
    RCC_init();      //初始化配置时钟频率为72MHZ
```

```
    LED_init();     //LED初始化配置
    Key_init();     //初始化控制按键的PA0端口

    while (1)
    {
        Delay(0xfffff);  //增加了防抖功能，如果你按下按键的时候会有抖动
        if(key_up)
            LEDON;          // 开灯
        else
            LEDOFF;         // 关灯

        if(KEY0==0)
            key_up = !key_up; //取反
    }
}
```

代码Delay(0xfffff)就是我们增加的防抖功能，如果你按下按键的时候会有抖动，我们增加一定时间的时间再来判断按键是否按下来，这样在一定程度上可以起到消抖的作用，但是这里还有一个BUG就是如果长时间按着按键不动，while循环里面就会运行多次灯点亮程序，如果是计数器的话，这个就不准了，按一次键，就计算了许多次数，大家可以尝试一下如何去解决这个问题，按一次键就只亮一次，无论一次按多长时间。

5.6 串口通信的收与发

5.6.1 串口通信

51 与 STM32 的串口通信原理都是相同的，基础知识建议到前面的 51 串口章节看看，这里直接用例程来引入和分析 STM32 的串口收发。

5.6.2 例程01 最简单串口打印$字符

1. 例程简介：

用一根杜邦线将 JP16 上的 P25 引脚和 JP19 上的任意一个灯的引脚连起来如图 5-62 所示。

STM32 的 GPIOA 端口的 PA9 和 PA10 位，即串口 1；设置 PA9 为 TX 输出模式，复用功能推挽输出模式；设置 PA10 为 RX 输入模式，模拟输入模式；对超级终端打印输出字符”$”符号，灯一亮一灭，PL2303HX 芯片与 ARM 芯片连接的原理图如图 5-60 所示：

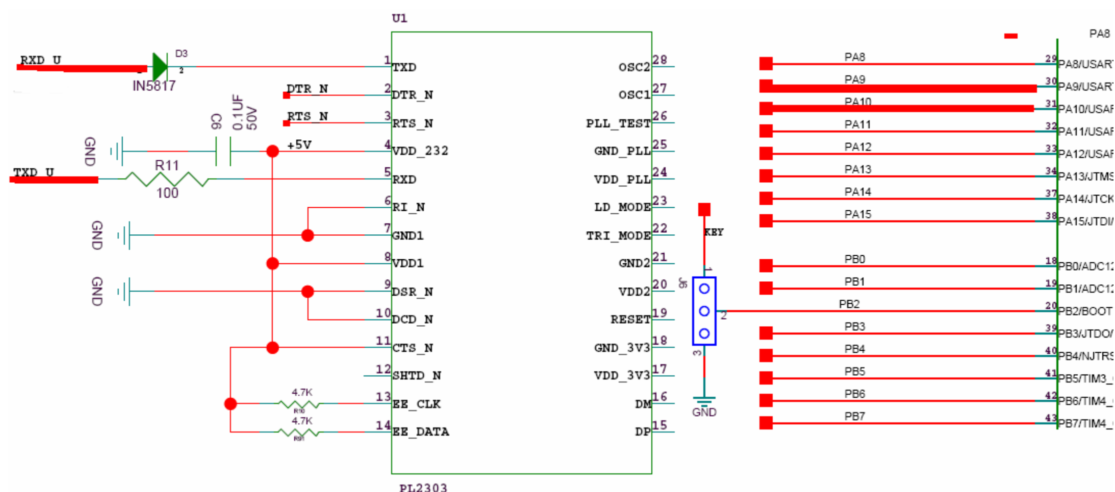


图 5-60 串口通信实验相关原理图

插入 USB 口，即可模拟成串口，具体硬件电路这里不细说。

2. 调试说明：

1） 打开开始菜单->程序->附件->通讯->超级终端，如图 6-36 所示：

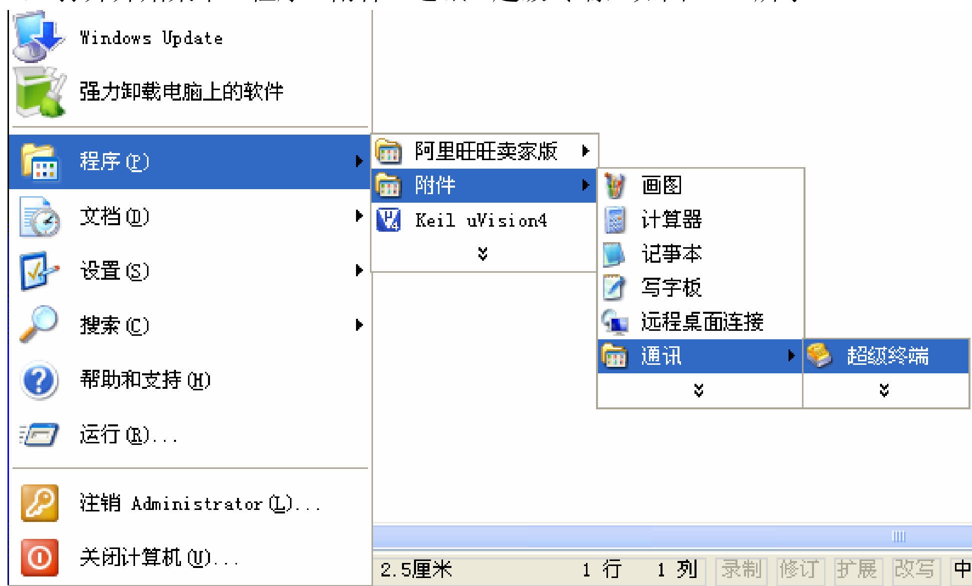


图 6-61 打开串口通信工具

2） 输入“STM32 神舟系列开发板”如图 5-62 所示：

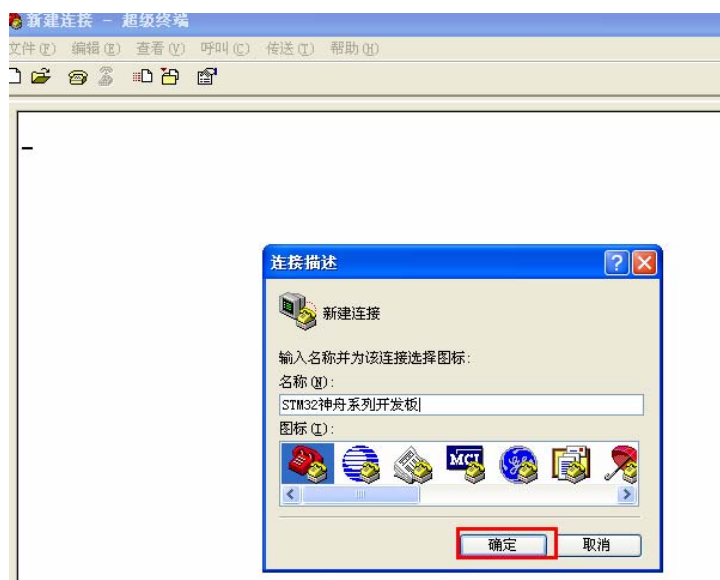


图 5-62 建立一个通信并取名

3） 将 STM32 神舟 51+ARM 的 USB 口与电脑的 USB 相连，然后打开设备管理器，查看是否识别到了一个 USB 转串口设备，我这里的是 COM4 接口如图 5-63 所示

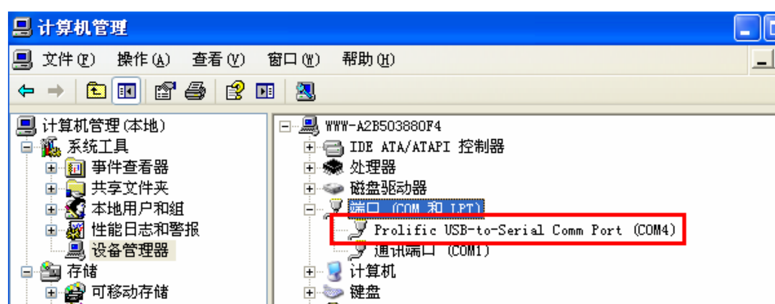


图 5-63 查看端口

4） 紧接着，输入 COM4 接口，此处选择选择图 6-63 显示的端口，如图 5-64 所示

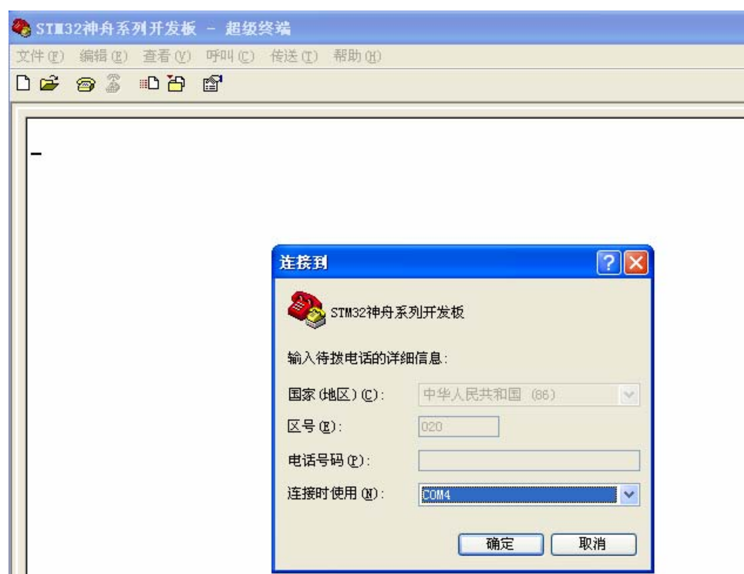


图 5-64 选择连接的端口

5） 波特率例程代码中设置的是 115200，数据流控制是无，选择完毕，点确定按钮，如图 5-65 所示



图 5-65 配置串口终端参数

6） 最后把例程程序下载到开发板里，然后按一下开发板复位或者重新上电，就会打印出“$”的字符，一会打印一个，恭喜发财哦！如图 5-66 所示

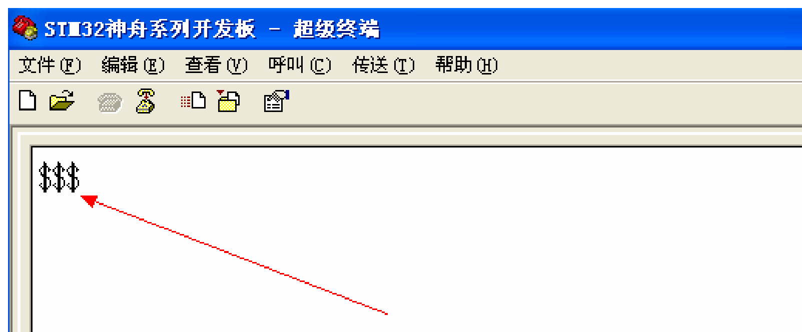


图 5-66 打印我们的数据信息

3. 关键代码：

```
int main(void)  //main 是程序入口
{
    RCC_init();        //时钟频率的配置
    LED_init();        //LED 初始化配置
    uart_init();       //串口接口初始化，这个部分是按 STM32 芯片手册的要求来做的，比较枯燥，细节感兴趣的朋友可以去研究下
    while (1)
    {
        USART1->DR = 0x24;    // 打印符号$，0x24 是 ASCII 码
        LEDON;                //点亮 LED 灯
        Delay(0xFFFFFF);      // 延时
        LEDOFF;               // 熄灭 LED 灯
        Delay(0xFFFFFF);      // 延时
    }
```

```
}
void uart_init()
{
      float USARTDIV;
      /*  因为 32 位的 USART_BRR 波特率设置寄存器只有低 16 位有效，所以这里我们
定义 16 位寄存器就足够了  */
      u16 USARTDIV_zhengshu;    //这里相当于 u16，无符号 16 位
      u16 USARTDIV_xiaoshu;      //这里相当于 u16，无符号 16 位

      RCC->APB2ENR|=1<<2;      //使能 PORTA 口时钟
      RCC->APB2ENR|=1<<14;    //使能串口时钟

      GPIOA->CRH&=0XFFFFF00F;
      GPIOA->CRH|=0X000008B0;      //IO 状态设置

      USARTDIV = (float)(72*1000000)/(115200*16);
      USARTDIV_zhengshu = USARTDIV;

      USARTDIV_xiaoshu = (USARTDIV - USARTDIV_zhengshu)* 16;
      USARTDIV_zhengshu <<=4;
      USARTDIV_zhengshu += USARTDIV_xiaoshu;

      RCC->APB2RSTR|=1<<14;          //复位串口 1
      RCC->APB2RSTR&=~(1<<14);    //停止复位

      USART1->BRR =    USARTDIV_zhengshu;
      USART1->CR1|=0X200C;        //1 位停止,无校验位.
}
```

代码详细分析：

（1）RCC_init();这个函数是负责时钟频率的配置，这里默认配置为 72MHZ，前面章节有详细分析，这里简化，有疑问的可以翻看前面的章节细节。

（2）LED_init()这个函数是初始化 LED 的配置，这个是附带的，如果串口无法正常打印，只要程序能运行，那 LED 灯就会进行闪烁。

（3）uart_init()这个函数负责串口的初始化，这个部分是按 STM32 芯片手册的要求来做的，比较枯燥，细节感兴趣的朋友可以继续往下看，不感兴趣的可以跳过这一节，这个函数要把波特率初始化为 115200，下面我们仔细分析一下代码：

（3-1）初始化一下波特率的值，一个是波特率的整数部分，一个是波特率的小数部分

```
u16 USARTDIV_zhengshu;    //这里相当于 u16，无符号 16 位，波特率的整数部分
u16 USARTDIV_xiaoshu;      //这里相当于 u16，无符号 16 位，波特率的小数部分
```

（3-2）初始化串口的时钟，再初始化 PA9 和 PA10 两个管脚的 GPIO 端口 A 的时钟。从原理图可以看到，USART1 的 TX 和 RX 就是 PA9 和 PA10，要使用串口不仅仅要初始化 GPIO 端口 A 的时钟，还要初始化串口的时钟，这里是需要注意的，如果点灯程序或者只做为普通的 GPIO 管脚使用，就不需要初始化串口的时钟。

串口时钟使能。串口作为 STM32 的一个外设，其时钟由外设始终使能寄存器控制，这里我们使用的串口 1 是在 APB2ENR 寄存器的第 14 位。这里需要注意的一点是，除了串口 1 的时钟使能在 APB2ENR 寄存器，其他串口的时钟使能位都在 APB1ENR。

```
RCC->APB2ENR|=1<<2;     //使能 PORTA 口时钟
RCC->APB2ENR|=1<<14;   //使能串口时钟
```

（3-3） 设置串口通信 RXD 和 TXD 的配置，一个管脚是接收就要设置为输入模式，一个管脚是输出，就设置成输出模式；寄存器 CRL 是设置 GPIO 的 0~7 位，CRH 是设置 GPIO 的 8~15 位，可以看到这里是设置 GPIOA 端口的 9 和 10 位，即 PA9 和 PA10 设置 PA9 为 TX 输出模式，复用功能推挽输出模式设置 PA10 为 RX 输入模式，模拟输入模式

```
GPIOA->CRH&=0XFFFFF00F;
GPIOA->CRH|=0X000008B0; //IO 状态设置
```

（3-4）设置波特率，在 CPU 是 72MHZ 的频率下，设置波特率为 115200；STM32 中波特率是如何计算的，首先看下文档：

$$\text{Tx / Rx 波特率} = \frac{f_{PCLKx}}{(16 * USARTDIV)}$$

f_{PCLKx}（x=1、2）是给外设的时钟（PCLK1 用于串口 2、3、4、5，PCLK2 用于串口 1）USARTDIV 是一个无符号的定点数，它的值可以有串口的 USART_BRR 寄存器值得到，寄存器的偏移地址为：0x08，复位地址为 0x0000，如表 5-43 和表 5-44 所示。而我们更关心的是如何从 USARTDIV 的值得到 USART_BRR 的值，因为一般我们知道的是波特率，和 PCLKx 的时钟，要求的就是 USART_BRR 的值。

表 5-43 USART_BRR 寄存器描述

注意：如果 TE 或 RE 被分别禁止，波特计数器停止计数															
地址偏移：0x08															
复位值：0x0000															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIV_Mantissa[11:0]												DIV_Fraction[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	w	rw	rw	rw

表 5-44 USART_BRR 寄存器（功能说明）

位 31:16	保留位，硬件强制为 0
位 15:0	**DIV_Mantissa[11:0]**：USARTDIV 的小数部分 这 12 位定义了 USART 分频器除法因子（USARTDIV）的小数部分。
位 3:0	**DIV_Fraction[3:0]**：USARTDIV 的整数部分 这 4 位定义了 USART 分配器除法因子（USARTDIV）的整数部分。

可以看到上图波特比率寄存器 USART_BRR 是低 16 位有效，高 16 位是闲置的，最低 4 位用来存放整数部分 DIV_Fraction，[15:4]这 12 位用来存放小数部分 DIV_Mantissa。高 16 位未使用。这里波特率的计算通过如下公式计算：

假设我们的串口 1 要设置为 115200 的波特率，而 PCLK2 的时钟为 72M。这样，我们根据上面的公式有：

$$波特率 = \frac{f_{PCLKx}}{(16 * USARTDIV)}$$

USARTDIV = f_{PCLKx} / 波特率*16 = (72*1000000)/(115200*16) = 39.0625

我们查看《STM32F10XX 参考手册》中的第 481 页的一个表，如表 5-45：

表 5-45 波特率误差计算

波特率		Fpclk = 36MHz			Fpclk = 72MHz		
序号	Kbps	实际	置于波特率寄存器中的值	误差%	实际	置于波特率寄存器中的值	误差%
1	2.4	2..400	937.5	0%	2.4	1875	0%
2	9.6	9.600	234.375	0%	9.6	468.75	0%
3	19.2	19.2	117.1875	0%	19.2	234.375	0%
4	57.6	57.6	39.0625	0%	57.6	78.125	0%
5	115.2	115.384	19.5	0.15%	115.2	39.0625	0%
6	230.4	230.769	9.75	0.16%	230.769	19.5	0.16%
7	460.8	461.538	4.875	0.16%	461.538	9.75	0.16%
8	921.6	923.076	2.4375	0.16%	923.076	4.875	0.16%
9	2250	2250	1	0%	2250	2	0%
10	4500	不可能	不可能	不可能	4500	1	0%

1．CPU 的时钟频率越低某一特定波特率的误差也越低

2．只有 USART1 使用 PCLK2（最高 72MHz）。其它 USART 使用 PCLK1（最高 36MHz）

USARTDIV 的值被设置为 39.0625，也就是 USART_BRR 寄存器

那么得到：

DIV_ Mantissa = 39 = 0x27;

DIV_ Fraction = 16*0.0625 = 1= 0x1;

这样，我们就得到了 USART1->BRR 的值为 0x271。只要设置串口 1 的 BRR 寄存器值为 0x271 就可以得到 115200 的波特率。

```
USARTDIV = (float)(72*1000000)/(115200*16);//算出 USARTDIV 的值
USARTDIV_zhengshu = USARTDIV;
/* 因为波特率设置寄存器是 USARTDIV 整数在 0~3 位，小数在 4~15 位乘以 16 是因为小数点后面是 4 位，将它右移过来取成整数 */
USARTDIV_xiaoshu = (USARTDIV - USARTDIV_zhengshu)* 16;
USARTDIV_zhengshu <<=4;
USARTDIV_zhengshu += USARTDIV_xiaoshu;
```

（3-5） 当 CPU 刚启动的时候一般都需要重新复位一下外设，确保该外设在正常供电

稳定后，能够稳定的工作可以看到复位一下之后就可以了，然后停止复位，让其开始正常工作。

串口复位。当外设出现异常的时候可以通过复位寄存器里面的对应位设置，实现该外设的复位，然后重新配置这个外设达到让其重新工作的目的。一般在系统刚开始配置外设的时候，都会先执行复位该外设的操作。串口 1 的复位是通过配置 APB2RSTR 寄存器的第 14 位来实现的。APB2RSTR 寄存器的各位描述如下表 5-46 所示：

表 5-46　APB2RSTR 寄存器描述

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADC3 RST	USART1 RST	TIM8 RST	SPI1 RST	TIM1 RST	ADC2 RST	ADC1 RST	IOPG RST	IOPF RST	IOPE RST	IOPD RST	IPPC RST	IOPB RST	IOPA RST	保留	AFIO RST
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	res	rw

从上图可知串口 1 的复位设置位在 APB2RSTR 的第 14 位。通过向该位写 1 复位串口 1，写 0 结束复位。其他串口的复位位在 APB1RSTR 里面。

```
RCC->APB2RSTR|=1<<14;      //复位串口 1
RCC->APB2RSTR&=~(1<<14);//停止复位
```

（3-6）把 115200 的波特率设置到 USART1->BRR 寄存器中，并且设置一下串口控制的寄存器 USART_CR1。STM32 的每个串口都有 3 个控制寄存器 USART_CR1~3，串口的很多配置都是通过这 3 个寄存器来设置的。这里我们只要用到 USART_CR1 就可以实现我们的功能了，这里主要设置该串口使能，正式启动这个串口功能该寄存器的描述在《STM32 参考手册》第 496 有更多的详细介绍，USART_CR1 的地址偏移值为 0x0C，复位值为 0x0000 0000，寄存器结构及说明分别如表 5-47 和表 5-48 所示。

表 5-47 USART_CR1 寄存器描述

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留		UE	M	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	IE	RE	RWU	SBK
res		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	w	rw	rw	rw

表 5-48 USART_CR1 寄存器功能描述描述

位	描述
位 31:14	保留位，硬件强制为 0
位 13	**UE**：USART 使能 当该位被清零，USART 的分频器和输出在当前字节传输完成后停止工作，以减少功耗。该位的置起和清零，是由软件操作的。 0：USART 分频器和输出被禁止； 1：USART 模块使能。

```
USART1->BRR =    USARTDIV_zhengshu;
USART1->CR1|=0X200C;    //1 位停止,无校验位.
```

（3-7）数据发送与接收。STM32 的发送与接收是通过数据寄存器 USART_DR 来实现的，这是一个双寄存器，包含了发送或接收的数据。由于它是由两个寄存器组成的，一个给发送用(TDR)，一个给接收用(RDR)，该寄存器兼具读和写的功能，寄存器描述如下：

USART_DR 的地址偏移值为 0x04，复位值为不确定值，寄存器结构及说明分别如表 5-49 和表 5-50 所示。

表5-49 USART_DR寄存器描述

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
保留							DR[8:0]								
							rw	rw	rw	rw	rw	rw	rw	rw	rw

USART_DR寄存器的各位功能配置如表5-50所示

表5-50 USART_DR功能配置

位	说明
位 31:9	保留位，硬件强制为 0
位 8:0	**DR[8:0]**：数据值 包含了发送或接收的数据，由于它是由两个寄存器组成的，一个给发送用（TDR），一个给接收用（RDR），该寄存器兼具读和写的功能，TDR 寄存器提供了内部总线和输出移位寄存器之间的并行接口（参见图 238）.RDR 寄存器提供了输入移位寄存器和内部总线之间的并行接口。 当使能校验位（USART_CR1 种 PCE 位被置位）进行发送时，写到 MSB 的值（根据数据的长度不同，MSB 是第 7 位或者第 8 位）会被后来的校验位该取代。 当使能校验位进行接收时，读到的 MSB 位是接收到校验位。

DR[8:0]为串口数据，可以看出，虽然是一个32位寄存器，但是只用了低9位（DR[8:0]），其他都是保留。

代码 USART1->DR = 0x24 是被用来打印符号$，0x24 是 ASCII 码请看下图 5-67 所示，通过把这个 0x24 输送给 USART_DR 寄存器后，就可以打印出$字符。

字符	十进制	十六进制	八进制	5/32
space	32	20	40	空格
!	33	21	41	无相关信息
"	34	22	42	无相关信息
#	35	23	43	无相关信息
$	36	24	44	无相关信息
%	37	25	45	无相关信息
&	38	26	46	无相关信息
'	39	27	47	无相关信息

图 5-67 ASCII 码

（4）进入 while(1)死循环，不停的让 LED 灯亮和灭，然后每次 LED 亮灭一次，就打印一个$字符，中间有一些延时，具体代码很简单。

5.6.3 例程02 单串口打印www.armjishu.com字符-初级

1. 例程简介：

   例程同上个一样，只是打印出www.armjishu.com这么多字符来，增加了一些ASCII码。

2. 调试说明：

   其他都与上个例程一样，下载进去后，打印出来的是如下图 5-68 所示：



图 5-68 串口例程终端打印数据

3. 关键代码：

   int main(void) //main 是程序入口

   {

   RCC_init(); //时钟频率的配置

   LED_init(); //LED 初始化配置

   uart_init(); //串口接口初始化，这个部分是按 STM32 芯片手册的要求来做的，比较枯燥，细节感兴趣的朋友可以去研究下

   while (1)

   {

   /* 通过查 ASCII 码表打印“www.armjishu.com”的 LOGO 每打印一个字符，都需要延时一下

   * 如果不加延时程序，就无法正确打印出，因为串口的缓冲数据需要一点时间才

```
送出去，所以需要等待一下 */
        USART1->DR = 0x77;   //w
        Delay(0xFFF);          // 延时

        USART1->DR = 0x77; //w
        Delay(0xFFF);          // 延时

        USART1->DR = 0x77; //w
        Delay(0xFFF);          // 延时

        USART1->DR = 0x2e;   //.
        Delay(0xFFF);              // 延时

        USART1->DR = 0x61;   //a
        Delay(0xFFF);              // 延时

        USART1->DR = 0x72; //r
        Delay(0xFFF);            // 延时

        USART1->DR = 0x6d;    //m
        Delay(0xFFF);              // 延时

        USART1->DR = 0x6a; //j
        Delay(0xFFF);            // 延时

        USART1->DR = 0x69; //i
        Delay(0xFFF);            // 延时

        USART1->DR = 0x73;   //s
        Delay(0xFFF);              // 延时

        USART1->DR = 0x68; //h
        Delay(0xFFF);            // 延时

        USART1->DR = 0x75; //u
        Delay(0xFFF);            // 延时

        USART1->DR = 0x2e;    //.
        Delay(0xFFF);              // 延时

        USART1->DR = 0x63;    //c
        Delay(0xFFF);            // 延时

        USART1->DR = 0x6f; //o
```

```
            Delay(0xFFF);           //  延时

            USART1->DR = 0x6d; // m
            Delay(0xFFF);           //  延时
            USART1->DR = 0x20; //  空格

            LEDON;                  //点亮 LED 灯
            Delay(0xFFFFFF);     //  延时
            LEDOFF;                 //  熄灭 LED 灯
            Delay(0xFFFFFF);     //  延时
        }
    }
```

代码分析：

可以看到，其实就是逐个打印出www.armjishu.com的ASCII码而已。

5.6.4 例程03 单串口打印www.armjishu.com字符-中级

1. 例程简介：

例程同上个一样，只是打印出www.armjishu.com这么多字符来，增加了一些ASCII码，主要体现代码撰写表达有区别。

2. 调试说明：

其他都与上个例程一样，下载进去后，打印出来的是如下图 5-70 所示



5-69 串口例程终端打印数据

3. 关键代码：

```
int main(void)   //main 是程序入口
{
        RCC_init();          //时钟频率的配置
        LED_init();          //LED 初始化配置
        uart_init();
        while (1)
        {
            USART1_Printf(" www.armjishu.com ");
            LEDON;              //点亮 LED 灯
            Delay(0xFFFFFF);     //  延时
            LEDOFF;                        //  熄灭 LED 灯
```

```
                Delay(0xFFFFFF);     //  延时
        }
    }
    void USART1_Printf(char *pch)
    {
        while(*pch != '\0')
        {
            USART_SendData(USART1,pch);
            pch++;
        }
    }
    void USART_SendData(USART_TypeDef* USARTx, char *Data)
    {
      USARTx->DR = *Data;
      Delay(0xFFF);
    }
```

代码分析：

（1） 可以看到，用 USART1_Printf(" www.armjishu.com ")这个函数逐个打印出 www.armjishu.com 的字符。

（2）USART1_Printf（）函数内部具体实现，就是把*pch指向这个www.armjishu.com其中一个单个字符，然后调用USART_SendData(USART1,pch)将这个字符往串口 1 发送输出，再用while(*pch != '\0')判断下一个字符是不是结束符，不是的话继续打印下一个字符。

（3）最后调用 USART_SendData(USART_TypeDef* USARTx, char *Data)这个函数，将字符数据的 USARTx->DR = *Data;给到 USART1_DR 寄存器，这里的 USART_SendData()函数中的*Data 等同于上个函数的 USART1_Printf ()函数中的*pch，而*pch 等同于 USART1_Printf()函数中的一个字符。

关于USART1_Printf(" www.armjishu.com ")，这个www.armjishu.com在USART1_Printf()函数中创造了一个内存空间，这个内存空间是在main函数开始调用USART1_Printf（）函数时分配的，而传入到USART1_Printf（）中的pch是为www.armjishu.com所分配的这个内存区域的一个指针的地址，然后继续把这个指针的地址传给USART_SendData()中的Data变量；换句话说，Data等于pch等于www.armjishu.com字符串内存区域的首地址。

5.6.5 例程04 单串口打印www.armjishu.com字符-高级

1. 例程简介：

例程同上个一样，只是打印出www.armjishu.com这么多字符来，主要体现代增加了一些串口传送校验。

2. 调试说明：

其他都与上个例程一样，下载进去后，打印出来的是如下图 5-70 所示



图 5-70 串口例程终端打印数据

3. 关键代码：

```
void USART1_Printf(char *pch)
{
    while(*pch != '\0')
    {
        USART_SendData(USART1,pch);
        while(USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET);
        USART_ClearFlag(USART1, USART_FLAG_TXE);
        pch++;
    }
}
void USART_SendData(USART_TypeDef* USARTx, char *Data)
{
    USARTx->DR = (*Data & (uint16_t)0x01FF);   // 这里的*Data 就是一个字符
    while((USARTx->SR&0x40) == 0);
}
```

代码分析：

（1） USARTx->DR = (*Data & (uint16_t)0x01FF) 这里的*Data 就是一个字符，ASCII 码是 0~127，并且 DR 是 0~8 个 bit 有效，可以看手册的 USART_DR 寄存器的描述，所以这里实际上我们只需要取二进制的前 9 位即可。

（2） while((USARTx->SR&0x40) == 0)，USART_SR 是状态寄存器的地址偏移值为 0x00，复位值为 0x00C0，寄存器结构及说明分别如表 5-51 和表 5-52 所示。

表 5-51USART_SR 状态寄存器描述

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留						CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NE	FE	PE
						rc w0	rc w0	r	rc w0	rc w0	r	r	r	r	r

USART_SR 寄存器第六位功能配置如下表 5-52：

表 5-52 USART_SR 状态寄存器第 6 位描述

位 6	**TC**：发送完成

	当包含有数据的一帧发送完成后，由硬件将该位置位。如果 USART_CR1 中的 TCIE 为 1，则产生中断。由软件序列清除该位（先读 USART_SR，然后写入 USART_DR）。TC 位也可以通过写入 0 来清除，只有在多缓存通讯中才推荐这种清除程序。 0：发送还未完成； 1：发送完成。

判断 TC 是不是已经发送完成，当包含有数据的一帧发送完成后，由硬件将该位置位，之前的代码是通过一定的延时，现在是判断寄存器，判断寄存器可以使得数据在第一时间发出之后，就能够使得开始进行下面的代码，比人工去延时的效率显著提高，所以算是比上个例程性能上的一种优化。

（3）当发送完一个数据后，while(USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET)查看一下寄存器说明如表 5-53 所示：

表 5-53 USART_SR 状态寄存器第 7 位描述

位 7	**TXE:**发送数据寄存器空 当 TDR 寄存器中的数据被硬件转移到移位寄存器的时候，该位被硬件置位。如果 USART_CR1 寄存器中的 TXEIE 为 1，则产生中断。对 USART_DR 的写操作，将该位清零。 0：数据还没有被转移到移位寄存器； 1：数据已经被转移到移位寄存器。 注意：单缓冲器传输中使用该位。

判断 USART _SR 寄存器当中的 TXE 是不是为 1，如果是为 1 才能结束这个 while 循环，当等于 1 的时候，数据已经被发送出去了，进入了移位寄存器。

（4）USART_ClearFlag(USART1, USART_FLAG_TXE)这句代码是我们查看《STM32F103 中文手册》获得 494 页 24.6.1 节 状态寄存器（USART_SR）章节，这里说的就是我们程序代码里的 USART_FLAG_TXE 这个标志位，等数据发送完毕后，再进行标志清空，方便进行下一次传输的时候有效使用。

5.6.6 例程05 USART-COM1串口接收与发送实验-初级版

1. 例程简介：

（1） 先将 J21 和 J22 的跳帽分别跳到 RXD+RS232 和 TXD+RS232。

（2） 用母对母交叉串口线将电脑（台式机）的串口和板子上的串口连接起来。

（3） 新建一个超级终端,台式机的预留串口一般默认为串口一。

例程主要实现在键盘上敲一个字符，输入这个字符到串口中，然后再通过超级终端打印出来。

2.调试说明：

1）下载完程序后，打开超级终端，敲入字符 d，可以看到整个超级终端不停的输出 d 这个字符，表示输入成功如图 5-71 所示。

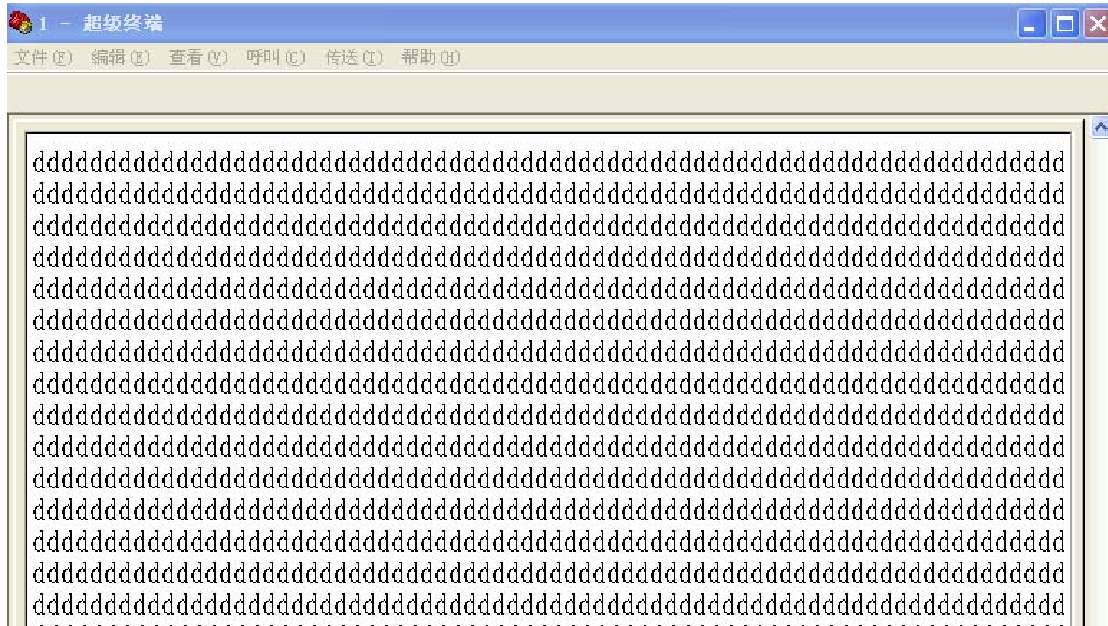


图 5-71 串口例程终端打印数据

2）重新按下复位按键，再次输入字符 a，可以看到界面满屏幕都输出整屏 a 如图 5-72 所示

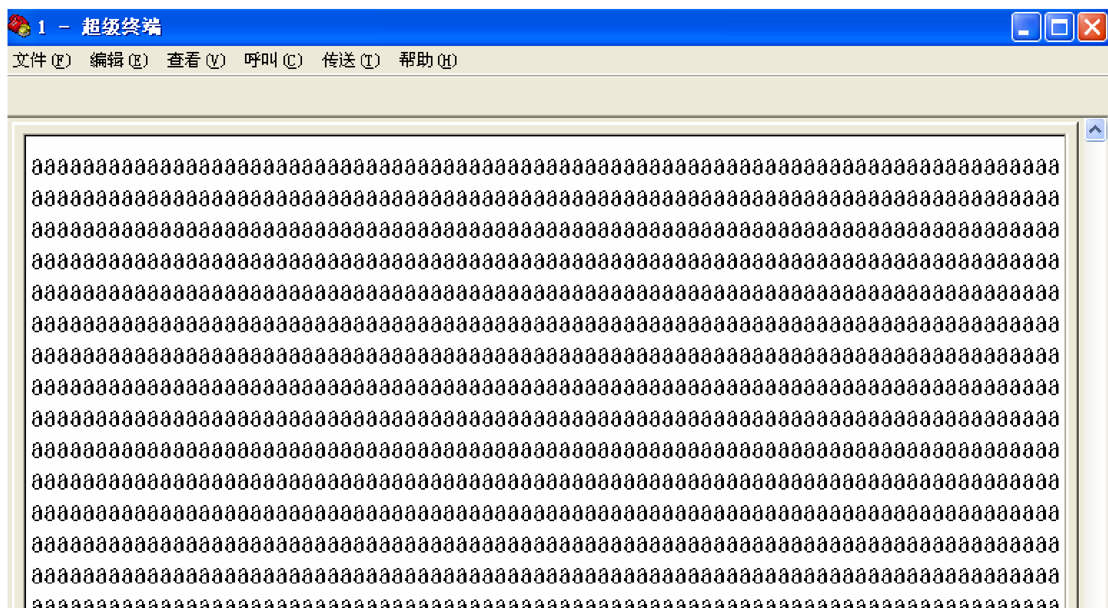


图 5-72 串口例程终端打印数据

4. 关键代码：

```
int main(void)
{
    uint8_t inputstr[CMD_STRING_SIZE];
    RCC_init();
    uart_init();
    GetInputString(inputstr);
}
```

```
void GetInputString (uint8_t * buffP)
{
    uint8_t c = 0;
    do
    {
        c = (uint8_t)USART1->DR;
        USART_SendData(USART1,c);
    }
    while (1);
}

void USART_SendData(USART_TypeDef* USARTx, uint8_t Data)
{
  USARTx->DR = (Data & (uint16_t)0x01FF);
}
```

代码分析：

（1） uint8_t inputstr[CMD_STRING_SIZE]代码声明一个数组，可以看到程序中变量 CMD_STRING_SIZE = 128，就是表示这个数组有 128 个成员，每个成员都是 uint8_t 类型的。

（2） 接下来就是时钟初始化函数 RCC_init()和串口初始化函数 uart_init()，之前都有详细介绍，这里就不做具体分析了。

（3） GetInputString(inputstr) 这句代码表示取 inputstr[]数组的初地址传入到 GetInputString()这个函数中，知道了 inputstr[]数组的首地址后，就可以在函数里操作和修改这个数组的内容。这里主要是熟悉一下指针和数组的一些基础概念，前面章节已经有详细分析。

（4） 可以看到数据寄存器 USART_DR，包含了发送或接收的数据。由于它是由两个寄存器组成的，一个给发送用(TDR)，一个给接收用(RDR)，该寄存器兼具读和写的功能。USART_DR 的地址偏移值为 0x04，复位值为不确定值，寄存器结构及说明分别如表 5-54 和表 5-55 所示：

表 5-54 数据寄存器 USART_DR 描述

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留							DR[8:0]								
							rw	rw	rw	rw	rw	rw	rw	rw	rw

表 5-55 数据寄存器 USART_DR（功能说明）

位	说明
位 31:9	保留位，硬件强制为 0
位 8:0	**DR[8:0]**：数据值 包含了发送或接收的数据，由于它是由两个寄存器组成的，一个给发送用（TDR），

	一个给接收用（RDR），该寄存器兼具读和写的功能，TDR 寄存器提供了内部总线和输出移位寄存器之间的并行接口（参见图 238）.RDR 寄存器提供了输入移位寄存器和内部总线之间的并行接口。 当使能校验位（USART_CR1 种 PCE 位被置位）进行发送时，写到 MSB 的值（根据数据的长度不同，MSB 是第 7 位或者第 8 位）会被后来的校验位该取代。 当使能校验位进行接收时，读到的 MSB 位是接收到校验位。

在 GetInputString（）函数中，用 c = (uint8_t)USART1->DR 这句代码来读取输入到超级终端的键盘字符，如果有输入，那么 c 就会得到输入的初值。

（5） 最后 USART_SendData(USART1,c)将 c 中的字符输出到超级终端上，细节请见代码。

5.6.7 例程06 USART-COM1串口接收与发送实验-中级版

1.例程简介：

例程主要实现在键盘上敲一个字符，输入这个字符到串口中，然后再通过超级终端打印出来，例程与上个不同的是“www.armjishu.com 键盘输入 2013 年:”的字符串，而这个字符串输入出来有一个人机交互界面的感觉，其他都与上个例程相同

2.调试说明：

1）下载完程序后，打开超级终端，按下复位按键，可以看到“www.armjishu.com 键盘输入 2013 年:”的字符串 ，然后敲入字符 d，可以看到整个超级终端不停的输出 d 这个字符，表示输入成功。

2）重新按下复位按键，可以再次看到“www.armjishu.com 键盘输入 2013 年:”的字符串，输入字符 a，可以看到界面满屏幕都输出整屏 a

3. 关键代码：

这里代码不做具体分析，之前有详细说

5.6.8 例程05 USART-COM1串口接收与发送实验-高级版

1. 例程简介：

例程主要实现在键盘上敲一个字符，输入这个字符到串口中，而之前的例程输出都是整屏不停的输出，这个例程加入了字符串部分代码的健壮，使得输入一个字符就是一个字符，绝对不会溢出或者数据丢失，或者重复输入。

2. 调试说明：

下载完程序后，打开超级终端，按下复位按键，可以看到“www.armjishu.com 键盘输入 2013 年:”的字符串 ，每敲一个字符，就会输出一个字符，不会有多余的字符出现。

3. 关键代码：

```
uint32_t SerialKeyPressed(uint8_t *key)
```

```
{
        if ( USART_GetFlagStatus(USART1, USART_FLAG_RXNE) != RESET)
        {
                *key = (uint8_t)USART1->DR;
                return 1;
        }
        else
        {
                return 0;
        }
}
uint8_t GetKey(void)
{
        uint8_t key = 0;

        while (1)
        {
                if (SerialKeyPressed((uint8_t*)&key)) break;
        }
        return key;
}
void GetInputString (uint8_t * buffP)
{
        uint32_t bytes_read = 0;
        uint8_t c = 0;
        do
        {
                c = GetKey();
                if (c == '\r')
                        break;
                if (c == '\b') /* Backspace */
                {
                        if (bytes_read > 0)
                        {
                                USART1_Printf("\b \b");
                                bytes_read --;
                        }
                        continue;
                }
                if (bytes_read >= (CMD_STRING_SIZE))
                {
                        USART1_Printf("Command string size overflow\r\n");
                        bytes_read = 0;
                        continue;
```

```
            }
            if (c >= 0x20 && c <= 0x7E)
            {
                buffP[bytes_read++] = c;
                SerialPutChar(c);
            }
        }
        while (1);
        USART1_Printf(("\n\r"));
        buffP[bytes_read] = '\0';
    }
```

代码分析：

（1） SerialKeyPressed(uint8_t *key)函数分析

状态寄存器 USART_SR 的地址偏移值为 0x00，复位值为 0x00C0，寄存器结构及说明分别如表 5-56 和表 5-57 所示。

表 5-56 状态寄存器 USART_SR 描述

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留						CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NE	FE	PE
						rc w0	rc w0	r	rc w0	rc w0	r	r	r	r	r

表 5-57 状态寄存器 USART_SR 第 5 位描述

位 5	**RXNE**：读数据寄存器非空 当 RDR 移位寄存器中的数据被转移到 USART_DR 寄存器中，该位被硬件置位。如果 USART_CR1 寄存器中的 RXNEIE 为 1，则产生中断。对 USART_DR 的读操作可以将该位清零。RXNE 位也可以通过写入 0 来清除，只有在多缓存通讯中才推荐这种清除程序。 0：数据没有收到； 1：收到数据，可以读出。

1）用语句 USART_GetFlagStatus(USART1, USART_FLAG_RXNE) != RESET 判断 USART_DR 寄存器中是否收到数据，如果收到了数据就可以读出。

2）*key = (uint8_t)USART1->DR; 第二步就是把数据取出来放到*key 中，*key 的此时的值就是键盘输入的字符的 ASCII 码的值。

（2） GetKey(void)函数在函数里分配了 key 的内存空间，然后调用了 SerialKeyPressed（）函数；其实这 GetKey 和 SerialKeyPressed 这两个函数在这个例程里可以合并，但是这样的写法有个好处就是，获取键盘的输入不一定是在串口中，也有可能是 can 的数据，或者 485，或者是网口等其他渠道的输入，用 GetKey()这个标准函数，再在里面调用具体的子函数，这样的好处就是增加了代码的可移

植性。

（3） GetInputString（）分析获得一个输入之后，就开始一系列判断，这个输入是不是空格，是不是换行符，如果是，就进行相对应的处理，这里值得注意的是，越详细，越复杂就表示你的输入这部分的代码越健壮，各种异常输入情况都考虑进去了；最后经过一连串的判断之后，合格的输入就调用 SerialPutChar()函数，把用户输入的内容输出到超级终端上。

5.6 更多ARM例程(42 个例程)包括详细代码分析

更多 ARM 相应的例程可通过北京航空航天大学出版社下载专区下载，如下表 5-58：

表 5-58 神舟 51 单片机 ARM 例程介绍

序号	例程功能
例程 01	单个 LED 点灯闪烁程序
例程 02	LED 双灯闪烁实验
例程 03	LED 三个灯同时亮同时灭
例程 04	-LED 流水灯程序
例程 05	STM32 芯片 32MHZ 频率下跑点灯程序
例程 06	STM32 芯片 40MHZ 频率下跑点灯程序
例程 07	-STM32 芯片 72MHZ 频率下跑点灯程序
例程 08	STM32 芯片按键点灯（无防抖）
例程 09	STM32 芯片按键点灯-增加了防抖的代码
例程 10	最简单串口打印$字符
例程 11	单串口打印 www.armjishu.com 字符-初级
例程 12	单串口打印 www.armjishu.com 字符-中级
例程 13	单串口打印 www.armjishu.com 字符-高级
例程 14	USART-COM1 串口接收与发送实验-初级版
例程 15	USART-COM1 串口接收与发送实验-中级版
例程 16	USART-COM1 串口接收与发送实验-高级版
例程 17	STM32 神舟开发板板载 LED 指示灯实验
例程 18	STM32 神舟开发板板载按键扫描实验
例程 19	STM32 神舟开发板 bitband 实验
例程 20	STM32 神舟开发板板载 EXTI 外部中断实验
例程 21	串口 printf 实验
例程 22	SysTick 系统滴答实验
例程 23	RCC 时钟配置实验
例程 24	独立看门狗实验
例程 25	RTC 实时时钟实验
例程 26	I2C EEPROM 访问实验
例程 27	STM32 神舟开发板 PS2 键盘实验
例程 28	STM32 神舟开发板 18B20 温测实验

例程 29	STM32 神舟开发板 NOKIA5110 液晶实验
例程 30	STM32 神舟开发板 LCD1602 实验
例程 31	ADC 模数转换实验
例程 32	2.4G 无线通信实验两个模块间通信
例程 33	STM32 神舟开发板 315M 无线模块
例程 34	USB 转串口实验
例程 35	TFT 彩色液晶屏只显示红色
例程 36	TFT 彩色液晶屏只显示蓝色
例程 37	TFT 彩色液晶屏显示英文字
例程 38	TFT 彩色液晶屏显示中文字
例程 39	TFT 彩色液晶屏显示中英文字
例程 40	ENC28J60 以太网程序（外接模块）
例程 41	UCOSII 操作系统之单任务运行
例程 42	UCOSII 操作系统之多任务运行

第六篇 嵌入式高手进阶之路

6.1 各种角色搭配组成

在从 51 到 ARM 的领域里，在一个实际的公司里，不仅仅只是一个研发人员，实际上所有的嵌入式产品，都需要如下的职位，您可能可以成为里面的一位，每个人员都是相互协作，相互配合，在大一点的公司，可能这些职位会分得更细；但是在小的公司里，可能一个人兼数职也有可能，主要职位如下：

6.1.1 产品经理

产品经理其实是一个最重要的职位，比如说乔布斯就是一名优秀的产品经理，虽然他有可能也是一名最强的技术总监，但是产品经理的烙印更加的深刻，因为用户需求很重要，如果把握不准用户的需求，设计出再好的产品也是白搭。

产品经理一般是由这个各行各业领域的专家来当任，熟悉特殊行业的一些规则，需求，当然也需要懂一点技术，但感觉国内很多产品经理对工艺和技术并不是特别在行，当然如果不懂技术，也是很难当好一个产品经理的，把握最新的需求和把握最新的技术自古就是两个方向的发展，可以两者都懂，但两者都很强大，这个就需要下一番大的功夫了。

假如您现在是苹果手机的产品经理，从人性角度出发如果有一款手持设备能够用一只手就能抓住，然后具备电脑性能，可以移动看新闻办公听音乐，那么站在这个角度出发，我们开始产品经理的思维模式。

因为要超薄，还有高处理性能，还要流畅操作系统，以及外围的软件工具生态系统也要全面，那么首先找芯片厂商、各种元器件厂商，触摸屏，TF 卡插槽等各厂家探讨可行性，是否有这么薄的器件，成本多少，如果定制是否可行，是否需要用更新的材料，这里涉及到的知识面非常广泛。

其次要找 CPU 厂商探讨是否将现有 CPU 的尺寸减少到一半，CPU 管脚能否减少四分

之一，芯片内部增加几个核，主频速度能否提高，参考设计如何，从预研到样片出厂的周期多长等；预研费用多少，如果研发成功，订单预期是多少，盈亏平衡点在哪里？

第三是进入到操作系统和生态，新功能的 CPU 芯片一定需要软件同步推出，否则就无法发挥 CPU 新功能；产品经理需要跟公司的操作系统部门沟通，要同步升级操作系统来支持新的 CPU；同时还要与软件工具生态系统沟通，推出新的软件工具来支持新 CPU 的新特性，例如打造软件免费下载的云市场。

总而言之，产品经理主要工作是满足一部分客户需求，客户有高端也有低端，客户群是有细分的，产品经理需要针对某个具体客户群做出合适的产品；产品经理必须对自己所处行业非常精通和了解，知道做一个这样的产品大概需要多少的预算和周期，也非常了解整个研究设计和生产流程，如何招聘人员如何打造和管理团队，灵活选用产品各组件和材质，比如现在透明钢化玻璃强度不比钢铁差，门窗产品经理就可以使用这种新材料推出的新门窗；一个优秀产品经理是非常难得的，实在是一将难求！产品经理一定是实战中战斗出来的，经验丰富，眼光独到沉稳，在目前 IT 领域和电子领域所有顶尖级优秀的产品经理全部都具有深厚的技术背景，所以菜鸟们需要慢慢熬哦。

6.1.2 技术总监

技术总监主要负责把握技术脉络，攻克技术难关，把握最前沿的技术动向，如果产品经理提出一个新产品的想法，就要由技术总监来把关，确定这个产品的思路是可以实现的；比如有个产品经理说，设计一个普通老百姓去月球上的飞行器吧，我想去月球；技术总监就会告诉你说那是不可能的，虽然中国已经实现了在月球步行，但那是举全国之力实现的，对个别公司来说，这无疑是一件特别难的事情，技术总监要把握的并不是能否实现这个技术的问题，而是要把握是否有足够的资金和实力去实现这个产品；假如一个普通公司来设计神舟火箭，那么他有足够的资金吗？他有足够的原材料提供吗？他有足够的科学家吗？如果社会上根本没有人懂这个，没人会做光有梦想是空谈，必须等待何时的时机，天时地利人和的时候才能做，所以技术总监主要把握产品实现的现实性，保证公司有足够的体力能够走到目的地。

在国内大部分科技公司里，可能有多个产品经理，而只有一个技术总监，技术总监可以为多个产品经理来服务，帮助这些产品经理来实现他们的产品；技术总监就像一颗树的树根，而树干和树叶都是公司技术人才，技术总监会分发养料给这些树干和叶子，技术总监到处寻找养料和化肥，找到合适自己树干和树叶需要的营养，源源不断的补给过去。

6.1.3 研发部经理

研发部经理主要负责的是一个产品研发进度，比如一个产品已经分配好了任务给下面 20 个软硬件研发人员，研发部经理就是盯着这些人的，处理研发过程中的一些难点，协调资源，如果产品分为 A、B、C 三个模块，那么谁做 A 模块，B 模块做着有困难时，从 C 模块调集几个高手来协助 B 模块恢复到正常的进度等等。

小公司研发经理兼做技术总监和产品经理，大公司分得比较细，研发部经理就像是一个持家人，要搭配好手上的资源，用到统筹原理，田忌赛马，跟员工谈心、讨论具体技术细节，撰写研发流程图，某个函数功能如何实现，某几个软件模块定义共同的全局变量，内存池如何划分，版本控制环境搭建，研发任务分配，研发 BUG 汇总调试，研发进度的把控，与公司高层沟通反馈本部门工作，起到一个承上启下的工作；技术人员如果从做技术开始就养成良好的习惯，逐渐积累管理经验，那么就可以逐渐往研发部经理方向转型。

6.1.4普通研发人员

普通研发人员就是干实际的事情的，当年雷军就是做了 N 年的金山研发人员，在第一线积累了丰富的经验，现在他做卓越网，凡客，以及现在的小米手机，都是因为他在第一线干过，所以他可以很好的把握一切实际的事情，当然技术和商业都是成功的保证，但普通研发人员就是一块基石，不可缺少。

读本书前面的那些内容就是普通研发人员需要懂的基础知识，如果没有两三年一线普通研发人员的经验，就很难说自己懂技术；最简单评判标准就是合作做几款产品，全程参与几款产品的研发经验比只做一个环节的技术工作知识面要来得全面很多。

6.1.5 售前工程师

产品种类非常多，具有各种特点，每个客户需求又各有区别，尤其对某些客户，客户本身是一个组织，或者公司，单位，部门，人数众多，对产品选型就需要售前工程师的介入；客户需要咨询一下产品性能，讨论和研究自己适合什么产品，采用什么样的方案组合，这样就需要售前工程师与客户沟通，售前工程师顾名思义，就是销售之前的工程师，是既懂技术又懂销售的一个职位。

这个职位现在已经被很多牛人占据，大多牛人都是从技术转型过来的，有些技术做疲倦了，也有因为随着年龄增大或者家庭需要照顾难以承担有压力的加班，身体也无法跟年轻比了，这些人就开始转型为售前工程师，虽然之前是技术背景不太懂销售，但随着日积月累，销售经验的也逐渐变强，当然客户也非常信赖这样的人，因为他多年一线的技术深厚积累，可以恰到好处的给客户好的建议，定制好的方案，所以现在这个职业的待遇薪水也是非常高的，甚至有的更进一步变成产品经理，技术总监或者是老总总监级的人也是比比皆是。

6.1.6 售后工程师

售后工程师顾名思义产品销售出去后有疑问，就找这个工程师来处理；这个职位主要是处理产品功能在实际使用过程中与客户需求衔接的问题，有时是产品具有这个功能但客户使用起来不方便或者是产品功能有缺陷，或者最直接的是产品质量存在问题。

当然这个职位不是单纯的维修工作，主要是偏向疑难杂症的处理，需要比较高的沟通技巧和对麻烦反应机敏以及的处理能力，因为毕竟是直接面对客户，当客户遇到困难时这个岗位的工作人员就是客户的天使，第一时间安抚他们的心情是最最重要的，然后立即处理好客户的问题，如果短时间处理不好也需要跟客户进行耐心的沟通，给出好的解决方案，所以这个职位也是非常锻炼人的。很多产品经理也大都尝试了这个职位，因为这样会让你知道在设计产品的时候，会考虑到客户使用反馈，从而使考虑问题更深度，更加长远。

6.1.7 销售

互联网和手机时代的到来，几乎颠覆了传统的销售，在电子和 IT 领域从事什么样的职业，我们从这个角度来分析一下。

目前来说 B2B 就是企业对企业的销售，有典型代表的阿里巴巴作为互联网平台进行销售；也有专业的销售人员通过专业展会，网络平台，行业协会等方式寻找到目标客户，直接通过网站，EMAIL，或者上门拜访的方式去为客户量身定做推销方案。例 1，客户某款产品一年出货量 100 万个，其中一颗主要芯片成本是 15 元，芯片公司销售协助客户修改方案改

用另外一颗成本只需要 7 元的方案，为客户节约成本 800 万元每年，芯片厂家销售不断在市场上寻找这样的客户，达到双赢；例 2，销售在某展会接了一个来自印尼的订单，印尼客户到展柜前订了 5000 台本公司的空气净化器，谈妥价格并收定金，确认交货期为 30 天；例 3，某公司需要设计一款工业级产品，销售推荐一款还在样品阶段的 CPU 芯片，公司经过两年设计出该产品并批量，刚好该 CPU 芯片也进入到了大批量出货阶段；例 4，某公司通过淘宝网销售电子元器件配单，销售人员负责把有优势的电子元器件包括 IC，电容电阻，二三极管，接插件以及其他相关产品上传到网页，客户看到自己需要的产品会通过旺旺聊天工具咨询，确认自己需要的产品，数量，谈好合理价格，最后淘宝下单并付款，这时销售安排发货；例 5，某公司发现很多烟酒饮料小卖店冰箱都是可乐公司，奶茶和冰红茶公司免费赠送，为了赠送冰箱不被挪作他用，需要研发出一款 GPS 全球定位加 GSM 短信发送二合一的小模块，该模块放入到冰箱隐蔽位置，装载物联卡，每隔 72 小时就会自动发位置信息到特定的云平台，这样公司总部可以方便看到冰箱准确位置是否与烟酒饮料小卖店一致。该模块成本为 50 元左右，销售价为 65 元；公司招聘销售人员去各冰箱公司推广，获得成功。例 6，华强北有一百多万种电子产品，没有任何一家能够全部备货压货，因为所需资金实在是太大，所以商家只需囤货 1-2 种器件，大家再彼此相互拿货配单就可以提供给客户，这好比是一个巨大的毛细孔血管组织，虽然密密麻麻，但配合得井井有条。例 7，小米模式，用一个论坛网站实时与消费者沟通，不断解决客户反馈问题，升级自己的产品，寻找最好的材料和最好的工厂代工，这就是所谓用户自己打造属于自己的产品……还有许多许多的例子。

也有 B2C 的销售，例如现在三大互联网腾讯，阿里，百度几乎都是 B2C 的销售；还有 C2C 的销售，例如直销，之前红火的微信卖面膜，以及各种代购服务都是个人用手机做的小生意，客户是针对个人。

目前科技发展非常迅速，各种销售模式层出不穷，本节所提的这些销售小例子或许过一两年就过时了，这方面本人也不是专家，只能给大家提供一点点启发，本人觉得，未来唯一不变的人性，所以不管如何变化，只要坚持挖掘和满足客户的需求，就是最好的销售，在客户有需求的前提下创造客户价值才是最靠谱的。

6.2 硬件专家之STM32 神舟团队 20 年工作经验心得总结

首先，如果你有幸看到这篇文章，千万不要试图在 2 个小时内阅读完，就算你 2 个小时阅读完，我相信你也不会理解里面讲解的精华之处，我相信，你应该将此文章，慢慢品尝，这绝对是一篇需要品尝 2~3 天，再结合自己过往的经验，加上自己的思考，我相信会对你不仅仅是技术能力，甚至包括整体的思维方式都会有一个非常大的提高。此篇文章摘抄于 www.armjishu.com 的坛主 jesse，如有需要转载，请注明作者，谢谢大家.

结合这篇文章，再结合 STM32 神舟系列开发板一些学习，可能会更加加深对嵌入式概念的理解。

我写这篇文章的目的，是用本人 20 年的嵌入式经验呈现给大家一副完整的产品，项目开发蓝图，用本人多年经的历总结了一些教训无私的分享给各位，希望各位今后能站在本人的肩膀之上，少走弯路，多为公司，为个人多做贡献，那我的愿望就达到了，也同时希望

能看到大家反馈和回复，留个脚印，留下你的见解和智慧，为后人乘凉打点基础，先在这谢谢各位了。

那么由此开始我们充满知识的旅程吧，最重要的一点，就是在一个产品或项目的开发过程中，如果没有明确的目标，那么成功将无从谈起，做任何事的第一步必须明确目标。

6.2.1 需求定义

需求定义用来描述产品的基本功能，对于公司来说，需求一般由该公司的市场销售部门或该公司的主要客户来制定；而对小公司或爱好者(就像 armjishu.com 里的爱好者一样)，技术人员可以自己负责定义需求，并撰写成文档；对于 STM32 神舟系列开发板来说，主要就是提供各种接口，为大家开发产品时提供借鉴！

通常需求定义是围绕以下几个因素而来：

1）系统的用途（定义需要系统实现的各种功能）

2）实际输入输出是何种方式实现的（为元器件的选型做参考）

3）系统是否需要操作界面（涉及软件层操作系统的选型）

其实对小型的嵌入式产品来说，定义需求是非常关键的，因为需求清楚了，就可以避免后续开发过程中出现的诸如随机存储器（RAM）容量不足或所选的 CPU 速度不能满足处理的需要等一系列问题。

下面举个简单的实际例子，供大家来参考：

系统描述：用于从化温泉的水泵换水系统（用 STM32 神舟 III 号开发板模拟实现）

电源输入：使用来自于变压器的 9V~12V 直流电

水泵功率：375W

1）使用单相交流电机，由机械电气进行控制

2）如果温泉池处于低水位，则输入开关闭合信号，以禁止水泵继续运行

3）用户可以自由设置水泵运行或关闭的时间长度

4）除了自动设置控制外，还需要提供一种人工装置来允许维护人员灵活控制水泵进行维修

5）水泵开启/关闭/人工干预的时间可以 30 分钟为单位，在 30 分钟到 23 小时的范围内进行调节

6）显示设备可以指示水泵的开关状态，剩余时间，以及水泵是否处于人工干预模式

7）具备监视低水位的功能，并显示在屏幕上

如果需要商用，那么除了上面给出的功能要求外，其设计文档中还要包括电磁干扰（EMI）和电磁兼容性（EMC）认证、安全认证以及使用环境（包括环境温度、湿度、盐雾腐蚀等）等方面的需求。

实际上，以上的需求确定之后，接下来就是要考虑选择一款合适的 CPU 来满足和实现系统的功能，那么我们就要将上述 7 点用户能够理解的需求转化成我们专业领域的需求，转化如下，大家可以参考一下：

a.处理或更新输入输出信号的速率究竟需要多快？

解释：目前嵌入式处理器的主频一般都在几十兆到几百兆不等，单片机的主频一般是几十兆，STM32 神舟系列开发板的 CPU 都是 72MHZ，有的 ARM9，ARM11 处理器可以到几百兆；我们主要看这个产品是否需要对大量数据进行处理，或是否需要对缓冲区进行频繁操作，是否有类似的占用 CPU 资料的工作要做，这就决定我们要选择一款合适的处理器来让该产品得到最佳的性能。

b.是否可使用单片集成电路（专用 IC）或 FPGA 来完成数据处理？

解释：如果可以的话，就不一定要选择处理器来做，用这些专业芯片就能替代

c.系统是否有大量的用户输入输出操作（如对开关和显示设备进行频繁操作）？

解释：如果有的话，要在处理器选型的时候考虑这些因素，选择一款能够满足以上要求的CPU.

d.系统与其他外部设备之间需要使用何种接口？

解释：这也是需要评估处理器的一个关键问题，选择具备这些接口功能的处理器会方便于我们的电路设计以及软件编程

e.设计完成后是否有可能需要进行改动，或在设计过程中系统需求是否可能出现变化？我们的设计是否能适应系统需求的变化？

解释:要避免选择的处理器刚好满足当前要求，这样当以后事务要求逐渐提高，处理器性能如果还有一定空间的话，那么就可以重用目前的产品；第二个就是要选择不会即将停产的芯片，很多处理器用得很广乏，可以借鉴的资料也很多，但是很可能这款芯片已经在市场上流行很长时间了，芯片厂商已经推出更新换代的替代品了，如果你选择了这款芯片，很可能 1，2 年后就买不到这款处理器芯片了，导致不得不重新选择新的处理器，重新设计产品，这样

的既耗费时间，金钱，更消耗人力，延误市场的战机。

与日常生活中的大多数事务一样，设计一个嵌入式产品的过程也必须从确定目标开始，对生产的产品进行明确定义。对产品进行定义主要是对产品是什么和能有什么功能进行描述，其次是在我们的整个开发过程中，应该要撰写一些开发文档，大概的框架的如下：

1）产品需求文档：描述产品的特性

2）功能需求文档：描述产品必须具备的功能

3）工程说明文档：描述系统实现的方法和满足需求的手段

4）硬件说明文档：对有关硬件进行描述

5）软件或固件说明文档：描述特定处理器下设计微程序以及固件的方法

6）测试说明文档：描述必须测试的项目和验证系统正常运行的方法

6.2.2 处理器的选择之IO管脚数量篇

多数处理器都是使用内存和外部管脚来控制输入输出设备的，通常处理器都会有内置 ROM 和 RAM 的，如果内置的内存就已经满足需要，那么处理器就可以节省产生引用外部存储器信号的引脚，这样处理器可为输入输出提供较多的设备管脚（某些处理器支持外部 RAM 或 ROM 的使用，但对外部存储器进行访问时，处理器一般需要占用 8 条到 10 条 I/O 管脚）。

还有，有些处理器带有专用的内部定时时钟，这类时钟也需要使用一个端口管脚来实现某些定时功能；某些处理器中还具有漏极输出和高电流输出能力，可以方便的直接驱动继电器或电磁铁线圈，而不再需要额外驱动硬件的支持。

当对处理器 I/O 管脚进行计数时，我们一定要把使用处理器内部功能（如串行接口和定时器等）时限制使用的某些管脚考虑在内。

6.2.3处理器的选择之接口需求篇

嵌入式处理器的主要功能是与应用环境中的硬件进行交互操作，这不仅需要外部硬件对接口具有实时处理能力，而且还要求处理器必须以足够快的速度对接口数据进行有效处理。

举例来说，STM32 神舟系列开发板的 CPU 是 ST 公司出品的一款工业级微处理器，它基于 CORTEX M3 的核心，处理主频可达 72MHZ，同时处理器内部配置了 USB、SPI、IIC 等接口，像 STM32 神舟 IV 号的 107 处理器还支持 Ethernet 等输出接口，其目的是更方便的利用这些接口开发出嵌入式产品。

需要注意的是，由于许多处理器具有的局限性没有在处理器技术资料中给予足够的说明，因此一定要仔细阅读处理器的指标说明。例如，在阅读资料的过程中发现，该资料可能会说明其串行接口可以在最高波特率下工作，但仔细研究该处理器的指标数据时，可能会发现并非该串口接口的所有操作模式都可以在最大波特率下运行。

深入了解并明确接口要求的方法：可以自己动手编写一些程序来对接口进行实际测试，以确认某种处理器是否可以满足应用的要求；因为，确认某个处理器是否可以满足接口要求并非是一件简单的任务。

6.2.4处理器的选择之内存容量需求篇

决定内存容量的大小是嵌入式产品设计过程中的一个基本步骤，如果对所需内存容量估计过高，那么我们就有可能会选择成本较高的解决方案；反之，如果低估了所需内存容量，就有可能因系统需要重新设计而导致项目不能按时完工。

a.RAM 和 ROM 的区别：存储器分为随机存储器（RAM）和只读存储器（ROM）两种。其中 ROM 通常用来固化存储一些生产厂家写入的程序或数据，用于启动电脑和控制电脑的工作方式。而 RAM 则用来存取各种动态的输入输出数据、中间计算结果以及与外部存储器交换的数据和暂存数据。设备断电后，RAM 中存储的数据就会丢失。

b.随即存储器（RAM）的选择：RAM 容量的预测是比较直观的，我们只需把所有变量数目与所有内部缓冲区的容量以及先入先出（FIFO）队列长度和堆栈长度直接相加，就能得到所需 RAM 容量的总数。

如果所需内存容量超出这类处理器的寻址范围，那么只能通过增加外部 RAM 来满足需求；然而，增加外部 RAM 的同时将会占用一定数量的 I/O 管脚来对扩展内存进行寻址，这种扩展往往会影响到处理器来实现应用的初衷。

需要注意的一个问题是，某些微处理器限制 RAM 的使用，这种限制的目的是为了借用部分内存存储器作为内部寄存器组使用。除了以上因素外，所使用的开发语言也对所需 RAM 容量有一定的影响，某些效率较低的编译程序可能会占用大量宝贵的 RAM 空间。

c.只读存储器（ROM）的选择：系统所需 ROM 的大小应该是系统程序代码与所有基于 ROM 的数据表容量之和。预测所需 ROM 空间容量比较困难的部分是预测程序代码的长度，解决这类问题的方法只能是随着经验的逐步积累来提高预测精度。

然而，最重要的并不是精确计算程序的代码长度，而是要清楚地估算代码长度的上限。根据经验，如果 80%的 ROM 空间被代码占用的话，那么就太拥挤了，除非能确保系统需求不会有任何变化，否则至少要为可能发生的变化保留足够的备用 ROM 空间。

在多数情况下，我们可以试着在 ROM 中写入一部分程序代码，以便观察代码占用空间的情况，对于带有内部 ROM 的微处理器系统来说，系统程序都只能占用有限的程序存储器空间。

d.经验之谈：ROM 与 RAM 使用情况相类似，程序代码长度与所选用的开发语言有关。举例来说，使用汇编语言编制的程序要比使用 C 语言编制的程序占用少得多的空间。

对于追求低成本的小型系统来说，一般不提倡使用高级程序设计语言；这是因为虽然高级语言在使用、调试以及维护方面来的比较容易，但同时这类语言需要占用更多的内存空间和大量的处理器时钟周期。

如果开发语言选择不当，其后果可能是把一个简单、低成本的单片机系统变为一个需要使用配置若干兆字节 RAM 空间的 64 位嵌入式处理器系统。

6.2.5处理器的选择之中断数量篇

中断的主要用途是向中央处理器通报当前发生的某类特殊事件，这类事件包括诸如定时器超时事件、硬件引发的事件等。

需要强调的是，多数系统设计师经常过多地使用中断功能，实际上，中断的主要作用只是中断现行程序的执行，中断最适用于必须要求中央处理器立即提供服务的事件。

在需要设计和使用中断的情况下，一定要首先确认实际需要的中断数量，然后必须考虑到系统内部占用的中断资源，如果需要使用的中断资源超出了处理器可以接收的中断数量，我们就应借助于某些特殊手段来减少所需中断信号的数量。

6.2.6处理器的选择之实时处理篇

实时处理是一个涉及范围很广的题目，其主要内容与系统的处理速度有密切联系，实时事件是嵌入式微处理器需要关注的主要任务。

例如：处理器跟串口进行通信时，通常通过上层软件（为了保证实时性，进行任务切换的时间足够短），然后再占用处理器去执行从串口拿数据的任务，并且要保证处理器的速率比串口速率快，那么处理器可以以最快的速度反应并处理串口的相关的任务，这样就可以达到最大的实时性；

另一方面，如果处理器本身就内置了串口控制器、或DMA、或LCD的控制器等，那么它就可以保证直接使用这些处理器内置的接口去控制串口、液晶屏等对象，以达到最大的实时性能。

6.2.7处理器的选择之芯片厂商篇

选择一款新的处理器，很可能就要使用一个新的开发工具和开发环境，包括软件的编译环境等；对于开发日程安排比较紧张的项目来说，开发人员往往无法抽出专门的时间来研究，熟悉新的开发工具，从而也无法全面掌握开发工具的使用技巧。
并且，有的开发工具价格也比较昂贵，而且很可能只能从制造商那里购买，还有仿真工具也是需要付费的，这些对我们在选择一款处理器的时候，是都应该考虑进去的成本因素。

6.2.8处理器的选择之芯片速度篇

主要考虑几个细节问题：

1）处理器速度与处理器时钟之间的关系

例：单片机 8031 为例，由该处理器可以适应 12MHz 频率的输入时钟，因此就可以认为它是一个速度为 12MHz 的处理器了吗？不是，实际上，由于该处理器内部逻辑电路执行每条指令需要多种不同频率的时钟脉冲，因此该处理器内部时钟电路要对输入的 12MHz 时钟 12 分频处理；最终为处理器提供的只是 1MHz 主频。

有的时候，80MHz 主频的处理器（80MHz 输入时钟，80MHz 执行速度）要比 200MHz 主频的处理器（200MHz 输入时钟，50MHz 执行速度）执行速度要快得多。

2）处理器指令系统

如果不需要执行复杂数学运算的应用，那么 RISC 指令集的处理器要快；如果执行比较复杂的操作，则 CISC 指令集的处理器速度要更快。

3）芯片结构体系

现在有的芯片是将多个不同功能的核封装到一个芯片 IC 中，定制某种特定的功能，比如 DSP，其中包括用于实现数字解码、乘法运算的硬件乘法器和移相器等；然而，这类处理器也由其自身局限，往往在执行某些普通操作之前必须要使用额外的指令来把 RAM 中的数据放入内部寄存器，相比之下，一般处理器只允许对 RAM 中的数据进行直接访问。

6.2.9处理器的选择之只读存储器(ROM)选择篇

多数工程项目在其开发阶段一般使用可擦写可编程只读存储器（EPROM）或快速存储器（Flash Memory）；这类可擦写可重复写入存储器的主要优点是可多次使用。一旦产品研制完毕，就可以用一次写入设备（OTP）来取代 EPROM 存储器，一次性写入器件的外观与封装几乎与 EPROM 完全一样，惟一不同之处就是其表面没有擦出窗口，并且价格要比 EPROM 低很多。

但是，另外一种情况，如果该产品今后需要升级固件，或在线编程，那么我们还是应该选择可擦写可编程的存储器。

还有一种是非易失的存储器，例如制造一台电视机，就有可能需要该设备具有记忆上次观看最后一个频道的功能，即使在切断电源后，该频道信息也不会丢失。

总结：所以，根据不同的产品选择不同的存储器也是一门很讲究的学问。

6.2.10 处理器的选择之电源要求篇

在某些设计中方案中，电源根本不存在问题，对电源唯一的要求就是可以为电路正常供电；实际上，选择电源主要要考虑三个方面的问题：

1）要注意设计方案中是否对电源的供电方式有所限制，例如,是否像大多数家用电器那样需要使用屋内墙上的电源插座供电，或是是使用USB接口供电

2）看系统是否需要使用电池供电方式，如果这样，我们就要考虑选择那种对驱动电流要求不高的处理器，然后再为其选择合适的电池。

3）休眠电流：许多微处理器都支持低功率运行模式，在这种模式下，系统的CPU处理器将处于休眠状态，同时所有外部设备的电源供电都被暂时切断，以便减少系统的电能消耗；某些微处理器在这种方式下需要的维持电流极小,但也有一些微处理器在这种方式下并不能节省多少功率；不管怎样，我们都要对系统在节点模式下的工作时间有一个估测，以便对具体情况选择使用的电池。
总之，无论哪种情况，我们都要对系统需要的供电总功率做到心中有数。

6.2.11 处理器的选择之设备工作环境要求篇

环境要求主要内容是考虑温度，湿度等；如果系统必须在温度范围较大的环境下运行，诸如用于军事设备或汽车的控制系统，那么处理器可选择的范围就要小得多；

并且由于大范围温度变化的设备通常比较昂贵,因此在设计过程中就不能再根据一般工业级器件的价格来制定预算。

6.2.12 处理器的选择之芯片寿命篇

如果我们的产品是 stm32 神舟开发板，在一般情况下，可以不必考虑在用户现场对 stm32 神舟开发板程序进行修改的问题，也不用为是否可以得到设备备件而着急，这是因为stm32神舟开发板是一种学习型的消费产品，仅仅只是一款开发板而已。

换句话说，如果我们的产品是价值几万块的工业设备并且需要常年不断地运行，那么我们在产品设计过程中就必须从长计议了：
a.首先，我们需要选择一种处理器或存储体系结构都可以升级的器件

b.考虑到程序升级的可能，我们还要选择较大容量的内存
c.最后要注意的则是所选处理器是否可以长期供货，这一点的重要性远远大于处理器的价格

除了上面的考虑之外，使用周期成本也是在设计之初要考虑的因素。总的来说，生产的部件越多，则可以接受的前期开发成本也就越大。如果产品是 mp3，我们可能会选择一个低价微处理器，同时投入一大笔钱来开发控制 mp3 的软件。
但如果我们的产品是价格昂贵的工业用设备，那么在产品的使用期内，该设备的销售量将只有几百台，毫无疑问，开发这种产品最重要的就是降低开发成本（降低开发成本而不是硬件成本！！！）；除此之外，工业产品的成本也不像家用电器或消费电子产品那么敏感。综上所述，开发工业产品当然要选择一种便于进行开发并且有助于缩短开发过程的处理器。

6.2.13 处理器的选择之资料获取篇

如果该款处理器在市场上已经用得很广了，那么我们可以获取更多的相关资料，观察人家的产品是如何使用处理器的，也能在网络上找到不少的相关的设计资料以及相关技术主题，这样就进一步降低了技术门槛，确保了使用该处理器做产品可行性，减低了风险；例如 STM32 神舟 IV 号开发板就有针对该板子有个 700 多页的手册文档，如果我们选择 STM32 芯片来开发产品的话，借助详细资料开发起来就轻松了，达到事半功倍的效果。

反之，如果是厂商全新推出的处理器，因为市场上还没有可以借鉴的产品，我们就只能从全英文的芯片手册开始阅读，了解这款芯片，这样开发周期不仅变长，而且不可预知的风险也很大。

6.2.14 开发成本的预测和估计

大多数项目或产品都有专人负责预测整个过程的开发成本，对于任何项目来说，其开发成本主要包括人力和材料开销。

预测开发成本在很大程度上需要根据经验，这也是为什么大型公司一般指定有经验的高级工程师来完成这一任务的原因，除了人力和材料的开销之外，总结下来，还有以下的开销：

1）人力成本（开发人员、管理人员、销售人员、其他行政等辅助人员）的开销
2）材料（硬件物料和损耗，有时候需要投几次 PCB 版才把产品稳定下来）的开销
3）开发系统和开发工具软件的开销
4）硬件工具的开销（例如示波器、仿真器等）
对于整个项目来说，上述的开销将直接可能导致产品成本增加，其中人力成本最为关键，尤其是在中国，呵呵。

6.2.15 产品开发设计文档之硬件文档撰写思路

1）首先是需求定义或产品规格：

如果这些是产品最终目标的话，那么产品对硬件和软件的要求就是技术方案的最终目标；对硬件和软件的要求是从定义用户界面和系统功能开始的。

2）其次，根据需求，系统整体定义文档中给出硬件接口的具体定义：

定义硬件最有效的方法是从需求开始描述，由于硬件必须支持系统定义的所有功能，因此硬件定义是与系统说明不可分割的；

例如,我们设计一个定时器（事先需求说明定时器不能与个人电脑连接，故无法使用 CRT 显示时间），我们只有两种选择：一种是使用发光二极管（LED），另一种是使用液晶显示器件（LCD）；尽管 LCD 的显示效果比较好，但考虑到定时器要常年位于户外，并且早期 LCD 显示器不能在低温下工作，最终还是选择 LED 设备（这整个过程描述了我们硬件选型时的一个思路，这个是密切跟需求挂钩的）

3）一旦完成了系统整体说明文档，就开始进行系统设计：

首先要对硬件说明的内容进行细化，包括添加能让工程师理解的设计意图，以及软件工程师围绕硬件进行程序设计时需要使用的硬件信息等。

完成硬件电路板说明文档后，我们还要在该文档中增加一个用来描述系统的原始要求的前言部分，包括说明方案的设计思路和方法，除此之外，还要附上软件工程师用来对硬件进行控制所需的各类信息，这类信息主要包括如下内容（软件工程所需信息）：

-----内存和 I/O 端口地址（如果需要，还可以提供内存映射图）

-----可用内存容量

-----状态寄存器每一位的定义

-----每个端口管脚的用途

-----外部设备的驱动方法（例如，说明输入定时器电路的时钟频率等）

-----其他有管软件人员设计程序需要了解的信息

对于比较复杂的系统来说，硬件文档中经常使用两个独立的部分来进行说明；其第一部分用来描述硬件指标和工作原理，第二部分则主要为软件人员提供程序设计需要的信息。

6.2.16 产品开发设计文档之软件文档撰写思路

1）软件文档与硬件文档的组织方法类似，软件要求文档的主要内容则是定义软件要实现的功能；一种是在简单项目设计过程中，软件定义也可以只对一种电路板使用的软件给予描述；对较复杂的项目来说，由于参与这种项目的软件人员分别负责设计驱动不同硬件部分的代码（同一电路板），因此每个软件人员可能会为自己的设计代码指定不同的定义，这类软件说明需要提供下列的内容：

-----论述包括需求定义、工程指标、硬件参数等实施项目需要的内容

-----说明软件之间、处理器之间或处理器与其内部器件之间使用的通信协议：其内容应包括对缓冲区接口机制、命令/应答协议、信号控制等协议的具体说明。

-----借助流程图、伪代码或者其他可能的方法来描述软件的实现方法和过程

2）软件与硬件所考虑的不同之处（此经验方便技术总监或其他相关管理者参考，因为无论是多高深的技术管理者，要么是硬件出身，要么是软件出身，要么就是非技术出身,armjishu.com 里面有少数软硬件都精通的高手）

a. 软件的灵活性远远大于硬件，要让软件人员搞清楚某个软件的内部格式是非常困难的任务，解决的办法：详细定义其他程序员需要了解的编程接口具体内容，以及其他工程人员在实施开发项目过程中需要使用的技术细节信息。

b. 软件工程师只有在收到硬件说明文档后，才有可能知道如何对系统硬件进行操作；而硬件人员一般不需要了解软件程序的技术细节。

c. 由于软件易于更改，因此程序内容经常会按销售人员提供的要求发生变更，在某些情况下，软件文档的内容无法及时反映程序的最新变化。

d. 软件经常是工程项目最后完成的部分，因此其文档也经常因时间不够而欠缺完整。实际上，软件文档是否详细、完整，在某种程度上是与公司或客户的要求有关的。例如，军事或国家工程一般要求开发商就其所有软件实现的功能提供全面详细的文档

e. 有个潜规则，对软件的要求越复杂，则需求的正确可能性就越小，这个是经验之谈了，我们需要把准需求这个准绳来做文章，而不是陷入个人主义以及对软件要求而凭空发挥自己不切实际的想象。

f. 我们可以先硬件设计，接着围绕该硬件编制软件。虽然实际系统的实现过程可能是软硬件并行开发，但软件人员基本上也是围绕着已经实现的硬件来进行程序设计的；对于更为复杂的系统来说，开发过程可能会出现重复。

例如，某个项目的硬件工程师和软件工程师可能会坐下来开会，共同决定使用哪种硬件来实现某种功能；软件人员可能提出需要为数据缓冲区口冲内存容量，也可能要求提供某种外部设备接口，以便充分利用现成接口程序提供的各种驱动代码。

总的来说，必须在提高软件开发效率与硬件系统的复杂性与成本之间进行权衡.

6.2.17 嵌入式高手对技术的理解（含辛茹苦这么多年的精华体验）

有很多人认为：嵌入式系统性能的核心因素是软件功能，其实，如果按照这种逻辑，系统设计中存在的问题就应由软件人员来负责;其实这个观点实际上反映了设计嵌入式产品时如何考虑划分硬件和软件各自应实现的功能，也就是这个功能是软件实现，还是考虑用硬件来实现（硬件实现：需要购买处理该功能的硬件芯片，从而增加成本；软件实现：无需增加硬件成本，但会占用处理器以及内存的资源，这是 armjishu.com 的专家们体会到的)。

例如：我们在这里设计的基于 STM32 的神舟 II 号开发板产品，我们可以使用专业的解码芯片来负责mp3音乐文件的解码和播放功能，也可以使用另一种方法来解码mp3语音文件，让 ARM 处理器利用软件控制寄存器来驱动耳机或音响，处理器通过对 mp3 语音文件解码之后再将解码后的数据流按照一定协议格式送给音频输出的硬件接口进行播放。

优点：这种方案在硬件方面节省了一个器件，降低了成本，并且该功能还方便调试（因为是软件实现的)。
缺点：从另一个角度来看，虽然节省了一块语音解码芯片，但同时要在三个方面增加成本。
首先，要在程序中增加语音协议解码的代码；
其次，可能要把增加 ROM 来存放语音解码的协议，这样可以增加速度；
最后，运行该程序将占用处理器的时间和资源。

其实，话又说回来，对于本案例来说，上述成本的节约并不会引发任何问题，包括驱动程序增加也只需少量的，我们讨论这个 mp3 产品的案例的目的在于说明如何对软件硬件的

功能进行合理划分。

总的来说，交给软件实现的功能越多，则产品的成本就越低，当然这就要处理器必须有足够的处理速度和内存空间来实现设计指定的功能;常言说得好,天下没有免费的午餐;把功能分配给软件来实现，会增加软件的复杂性、开发时间、以及程序的调试时间；然而，随着处理器的处理能力的不断提高，可以预见，越来越多的功能将会由软件来实现。

虽然在软件中实现各种功能会增加开发成本，但如果把功能移植到硬件中实现，则会增加产品的成本，这类开销是在构造每个系统组件时不可避免的。在低成本设计方案中，增加任何额外的硬件都会对产品成本产生显著的影响，因此软硬件功能划分就是一个决定产品成本的大问题。在诸如大众消费产品这一类对成本非常敏感的设计方案中，一般都会把无法通过软件实现的功能排除在外的。

6.3 PCB设计建议

总的来说，必须在提高软件开发效率与硬件系统的复杂性与成本之间进行权衡.

6.3.1 PCB设计干扰的相关基础知识

EMI：电磁干扰（Electromagnetic Interference 简称 EMI），直译是电磁干扰。这是合成词，我们应该分别考虑"电磁"和"干扰"。是指电磁波与电子元件作用后而产生的干扰现象，有传导干扰和辐射干扰两种。传导干扰是指通过导电介质把一个电网络上的信号耦合(干扰)到另一个电网络。辐射干扰是指干扰源通过空间把其信号耦合(干扰)到另一个电网络，在高速 PCB 及系统设计中，高频信号线、集成电路的引脚、各类接插件等都可能成为具有天线特性的辐射干扰源，能发射电磁波并影响其他系统或本系统内其他子系统的正常工作。

所谓“干扰”,指设备受到干扰后性能降低以及对设备产生干扰的干扰源这二层意思。第一层意思如雷电使收音机产生杂音,摩托车在附近行驶后电视画面出现雪花,拿起电话后听到无线电声音等,这些可以简称其为与“BC I”“TV I”“Tel I”,这些缩写中都有相同的“I”（干扰）。

那么 EMI 标准和 EMI 检测是 EMI 的哪部分呢？理所当然是第二层含义,即干扰源,也包括受到干扰之前的电磁能量。

6.3.2 电磁干扰三要素

理论和实践的研究表明，不管复杂系统还是简单装置，任何一个电磁干扰的发生必须具备三个基本条件：首先应该具有骚扰源；其次有传播干扰能量的途径和通道；第三还必须有被干扰对象的响应。在电磁兼容性理论中把被干扰对象统称为敏感设备（或敏感器）。

因此电磁骚扰源、骚扰传播途径（或传输通道）和敏感设备称为电磁干扰三要素。

6.3.3 电磁骚扰源分类

1.1．一般说来电磁骚扰源分为两大类：自然骚扰源与人为骚扰源。

自然干扰源主要来源于大气层的天电噪声、地球外层空间的宇宙噪声。他们既是地球电磁环境的基本要素组成部分，同时又是对无线电通讯和空间技术造成干扰的干扰源。自然噪声会对人造卫星和宇宙飞船的运行产生干扰，也会对弹道导弹运载火箭的发射产生干扰。

人 为干扰源是有机电或其他人工装置产生电磁能量干扰，其中一部分是专门用来发射电磁能量的装置，如广播、电视、通信、雷达和导航等无线电设备，称为有意发射干扰源。另一部分是在完成自身功能的同时附带产生电磁能量的发射，如交通车辆、架空输电线、照明器具、电动机械、家用电器以及工业、医用射频设备等等。因此这部分又成为无意发射干扰源。

1.2．从电磁干扰属性来分，可以分为功能型干扰源和非功能性干扰源。

功能性干扰源系指设备实现功能过程中造成对其他设备的直接干扰；非功能性干扰源是指用电装置在实现自身功能的同时伴随产生或附加产生的副作用，如开关闭合或切断产生的电弧放电干扰。

1.3．从电磁干扰信号频谱宽度，可以分为宽带干扰源和窄带干扰源。

他们是相对于指定感受器的带宽大或小来加以区别的。干扰信号的带宽大于指定感受器带宽的成为当代干扰，反之称为窄带干扰源。

1.4．从干扰信号的频率范围来分

可以把干扰源分为工频与音频干扰源（50Hz 及其谐波）、甚低频干扰源（30Hz 以下）、载频干扰源（10kHz~300kHz）、射频及视频干扰源（300kHz）、微波干扰源（300MHz~100GHz）。

6.3.4 电磁骚扰传播途径

电磁干扰传播途径一般也分为两种：即传导耦合方式和辐射耦合方式。

任何电磁干扰的发生都必然存在干扰能量的传输和传输途径（或传输通道）。通常认为电磁干扰传输有两种方式：一种是传导传输方式；另一种是辐射传输方式。因此从被干扰的敏感器来看，干扰耦合可分为传导耦合和辐射耦合两大类。

传导传输必须在干扰源和敏感器之间有完整的电路连接，干扰信号沿着这个连接电路传递到敏感器，发生干扰现象。这个传输电路可包括导线，设备的导电构件、供电电源、公共阻抗、接地平板、电阻、电感、电容和互感元件等。

辐射传输是通过介质以电磁波的形式传播，干扰能量按电磁场的规律向周围空间发射。常见的辐射耦合由三种：1. 甲天线发射的电磁波被乙天线意外接受，称为天线对天线耦合；2. 空间电磁场经导线感应而耦合，称为场对线的耦合；3.两根平行导线之间的高频信号感应，称为线对线的感应耦合。

在实际工程中，两个设备之间发生干扰通常包含着许多种途径的耦合。正因为多种途径的耦合同时存在，反复交叉耦合，共同产生干扰，才使电磁干扰变得难以控制。

6.3.5 印制电路板

出于技术的考虑，最好使用独立的接地层（VSS）和专门独立的供电层（VDD）的多层印制电路板，这样能提供好的耦合性能和屏蔽效果。一般 4 层板以上的结构就可以满足了，中间的两层一层是地，一层是走供电层，最上面和最下面的两层走信号线；很多应用中，受经济条件限制不能使用这样的印制电路板，一般单层和两层的 PCB 电路板就需要更多细节去考虑一下地和供电走线的配置了，下面会进行进一步的详细分析。

6.3.6 器件位置

为了减少 PCB 上的交叉耦合（注：耦合是指两个或两个以上的电路元件或电网络的输入与输出之间存在紧密配合与相互影响，并通过相互作用从一侧向另一侧传输能量的现象），设计 PCB 时需要根据各自对 EMI 影响的不同，而把不同的电路分开。比如，大电流电路、低电压电路以及数字器件等；比如某 CPU 的晶振必须靠近该 CPU，电源芯片的滤波电容尽可能的靠近电源芯片管脚附近，这样效果都会好很多。

6.3.7 接地和供电（VSS，VDD）

每个模块（噪声电路、敏感度低的电路、数字电路）都应该单独接地，所有的地最终都应在一个点上连到一起。尽量避免或者减小回路的区域。为了减少供电回路的区域，电源应该尽量靠近地线，这是因为，供电回路就像个天线，成为 EMI 的发射器和接收器。PCB 上没有器件的区域，需要填充为地，以提供好的屏蔽效果（特别是对单层 PCB，尤其如此）。

既使在整个 PCB 板中的布线完成得都很好，但由于电源、 地线的考虑不周到而引起的干扰，会使产品的性能下降，有时甚至影响到产品的成功率。所以对电、地线的布线要认真对待，把电、地线所产生的噪音干扰降到最低限度，以保证产品的质量。

对每个从事电子产品设计的工程人员来说都明白地线与电源线之间噪音所产生的原因， 现只对降低式抑制噪音作以表述：众所周知的是在电源、地线之间加上去耦电容。

尽量加宽电源、地线宽度，最好是地线比电源线宽，它们的关系是：地线＞电源线＞信号线。

6.3.8 数字电路与模拟电路的共地处理

现在有许多 PCB 不再是单一功能电路（数字或模拟电路），而是由数字电路和模拟电路混合构成的。因此在布线时就需要考虑它们之间互相干扰问题，特别是地线上的噪音干扰。 数字电路的频率高，模拟电路的敏感度强。

对信号线来说，高频的信号线尽可能远离敏感的模拟电路器件。

对地线来说，整人 PCB 对外界只有一个结点，所以必须在 PCB 内部进行处理数、模共地的问题，而在板内部数字地和模拟地实际上是分开的它们之间互不相连，只是在 PCB 与外界连接的接口处（如插头等）。

数字地与模拟地有一点短接，请注意，只有一个连接点。也有在 PCB 上不共地的，这由系统设计来决定。在神舟开发板中，我们使用的 0 欧电阻隔离了一下，实际产品中，建议

使用磁珠进行隔离会比较好。

6.3.9 信号线布在电或地层上

像神舟王的核心板 STM32103,207,407,439 都是 4 层板，这个问题一般都会在多层设计的时候出现，在多层印制板布线时，由于在信号线层没有布完的线剩下已经不多，再多加层数就会造成浪费也会给生产增加一定的工作量，成本也相应增加了，为解决这个矛盾，可以考虑在电（地）层上进行布线。首先应考虑用电源层，其次才是地层。因为最好是保留地层的完整性。两层板不会出现这个问题。

6.3.10焊盘与产品良品率质量的关系

设计的焊盘大小和形状也与产品的质量有关系，举 2 个例来说明：

在大面积的接地（电）中，常用元器件的管脚与其连接，对连接管脚的处理需要进行综合的考虑，就电气性能而言，元件管脚的焊盘与铜面满接为好，但对元件的焊接装配就存在一些不良隐患，比如有时候焊盘过大，造成加工的时候加热时间要久一些，如果加热时间不够，就有可能造成虚焊。

如果焊盘过短也会有问题，很容易造成接触不良，所以说，焊盘的长短大小，都有一个适度的的大小，如果说小了短了，就会造成管脚接触不到位，焊盘面积与芯片管脚接触面过小，造成通信质量下降；并且焊盘过短和过小，将会造成贴片加工难度增加，有个调查很有意思，一批产品一次性加工出来良品率是 85%，经过摸索，这个产品被发现有个芯片总出现虚焊的情况，后来把这个芯片的焊盘加长一点点，再次批量加工，这颗芯片几乎都不会出现虚焊的情况，良品率一度上升至 98%。

6.3.11其他信号的注意事项

实际应用中，关注以下几点可以提高 EMC 性能：

● 那些受暂时的干扰会影响运行结果的信号(比如中断或者握手抖动信号,而不是 LED 命令之类的信号)。对于这些信号，信号线周围铺地，缩短走线距离，消除邻近的噪声和敏感的连线都可以提高 EMC 性能。 对于数字信号，为有效地区别 2 种逻辑状态，必须能够达到最佳可能的信号特性余量(译注：尽可能抬高逻辑’1’的高电平，拉低逻辑’0’的低电平)。推荐使用慢速施密特触发

器来消除寄生状态。

● 噪声信号(时钟等)。

● 敏感信号(高阻等)。

6.3.12未用到的I/O管脚

所有微控制器都为各种应用而设计，而通常的应用都不会用到所有的微控制器资源。为了提高 EMC 性能，不用的时钟、计数器或者 I/O 管脚，需要做相应处理，比如，I/O 端口应该被设置为’0’或’1’(对不用到的 I/O 引脚上拉或者下拉)；没有用到的模块应该禁止

或者“冻结”。

因为如果不冻结，有可能会出现异常情况，并且出现这些情况之后，还不方便查找问题，所以干脆将这些管脚规定好，这样方便使得产品稳定，并且万一出现问题就会比较稳定。

6.4 软件领域专家

本章通过简单介绍 STM32 库的各个文件及其关系，让读者建立 STM32 库的概念，看完后对库有个总体印象即可，在后期实际开发时接触了具体的库时，再回头看看这一章，相信你对 STM32 库又会有一个更深刻的认识。

6.4.1 STM32库函数到底是什么

STM32 的库函数是 ST 公司已经封装好一个软件封装库，也就是很多基础的代码，在开发产品的时候只需要直接调用这个 ST 库函数的函数接口就可以完成一系列工作；例如，你原来要自己烧饭洗衣服，现在 ST 库函数就好像一个助手，你只需要使唤一声，请帮我来一杯咖啡，就有一杯咖啡摆放在你的面前。

这个 STM32 库提供标准的接口，你在程序中引用这些标准的接口，使用接口所描述的功能，接口内部代码的具体实现由 ST 原厂的顶级工程师来负责撰写和更新维护。相当于使用 STM32 库函数这样一种模式，就使得原本是你一个人在写代码，变成了你与 ST 原厂研发团队合力成为了一个强大的虚拟团队了，这是一种工作效率的提升。

6.4.2 STM32库函数的好处

在 51 单片机的程序开发中，我们直接配置 51 单片机的寄存器，控制芯片的工作方式，如中断，定时器等。配置的时候，我们常常要查阅寄存器表，看用到哪些配置位，为了配置某功能，该置 1 还是置 0。这些都是很琐碎的、机械的工作，因为 51 单片机的软件相对来说较简单，而且资源很有限，所以可以直接配置寄存器的方式来开发。

STM32 库是由 ST 公司针对 STM32 提供的函数接口，即 API (Application Program Interface)，开发者可调用这些函数接口来配置 STM32 的寄存器，使开发人员得以脱离最底层的寄存器操作，有开发快速，易于阅读，维护成本低等优点。

当我们调用库的 API 的时候可以不用挖空心思去了解库底层的寄存器操作，就像当年我们学习 C 语言的时候，用 prinft()函数时只是学习它的使用格式，并没有去研究它的源码实现，如非必要，可以说是老死不相往来。

实际上，库是架设在寄存器与用户驱动层之间的代码，向下处理与寄存器直接相关的配置，向上为用户提供配置寄存器的接口如下图 6-1 所示。

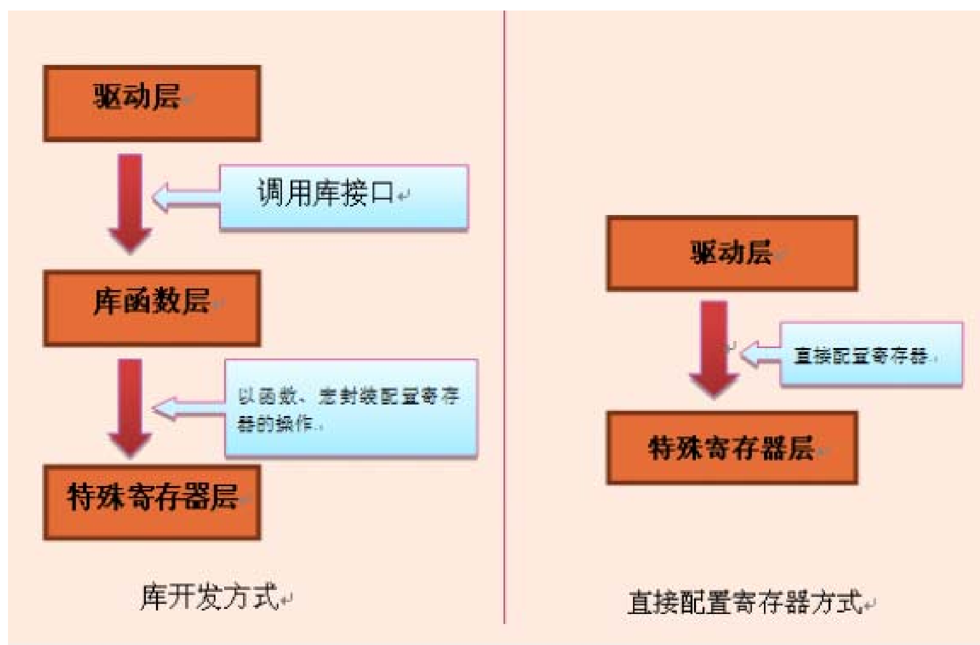


图 6-1　配置寄存器的接口

主要的好处有如下：

- **方便整个项目的移植**

  应用工程师只认库函数工程师提供的函数接口名称，去调用这个接口就可以，比如调用 open（）函数，就是打开一个什么功能，至于具体是什么功能，可以看库函数工程师提供的文档说明，这个函数的内部实现由库函数工程师来完成；应用工程师不用管，只需要调用就可以。

- 项目方便分工

  接口比较明朗了，库函数实现的人员被称作驱动研发人员，应用工程师负责上层应用；这样的话，一个研发团队，就已经被分成了两拨人员。

  下来继续分配，驱动研发人员，可以按照 CPU 的各个接口来分工，比如串口一个人做，以太网一个人做，CAN 由专人负责，485 又另外的同事负责这样子安排；应用工程师可以按照功能需求来进行分工。

- 项目研发速度加快

  因为软件架构清晰，所以会使得分工非常的明确，这样人数多了，并且管理还不会乱，大家都按照统一的框架来进行研发，这样项目的速度进展就会比较快。

- 项目方便维护

  各方面都清晰清楚之后，项目就方便维护了，驱动研发工程师有驱动的文档，有给应用工程师的接口文档，哪里出问题，就找哪块的负责人，每个人的精力都可以专注的研究一个部分，并且各自都是模块化，比如 open（）函数，应用工程师使用了驱动工程师提供的 open()函数来做程序，今后 open（）底层要修改，驱动工程师只修改底层的代码就可以，上层应用根本不需要动。

6.4.3 千人大项目如何分配工作

千人大项目有两种类型的项目，一种是类似 Windows 操作系统这样的大项目；另外一种类似腾讯百度阿里巴巴这类公司的大项目；这两种软件架构以及人员部署是不同的。下面分别讲解一下：

第一种类似 windows 操作系统项目，其实大多数项目产品研发都是用类似库函数的方式进行分解的，如果有 1000 个开发人员来负责开发板 Windows 操作系统，那么怎么做？一定是有一群人负责最底层的硬件级，寄存器的读写封装，包括显示器的点亮，图形刷写；然后另外一群人根据底层这群人提供的接口，同步做二次封装的开发。

软件项目越大，就要分多几层，比如硬件驱动层，普通驱动层，功能函数层，应用层等，每个层各自定义好接口，每个层都可以安排一个项目经理，相互协同互相协作的研发，普通驱动层调用硬件驱动层的元函数或者叫单功能函数；功能函数函数层偏向用户的行为模式，所以功能函数可能会组织多个普通驱动层的函数来组合成一个功能，一个用户能够理解的功能；最后是应用层的函数，应用层的函数可能会组合多个功能函数层，来完成自己复杂的应用，比如你在液晶屏上点一个按钮，就要通过 WIFI 无线传一个订单到服务器上，按下这个按钮之后，可能调用了液晶屏的触摸功能函数，调用了 CPU 与 WIFI 连接的功能函数与无线发送完毕反馈的函数，以及液晶屏显示发送成功的功能函数等一系列函数；只要架构分清楚，结构清晰了，就可以合适的分工，千人大项目就是这样来的；所以库函数的理念完全被广泛应用于各种实际的项目和产品中，因为这样才可以使得多人协同工作，才能做更大的项目产品。

另外一种类似腾讯百度阿里巴巴这类公司的大项目是如何研发和设计出来的呢？比如腾讯的 QQ 和微信，比如淘宝网，阿里巴巴网站，阿里旺旺，支付宝，百度搜索，百度云服务等每个功能块下面又有无数强大的软件服务，背后的软件代码相当庞大，它们是如何组织在一起的？是库函数吗？驱动层吗？不是的，这里要引入一个新的概念，叫做域名，实际上是域名背后的服务器提供了这项服务，例如淘宝网有搜索功能，搜索数据发到搜索服务器上，取回数据再返回给客户端；淘宝网系统内发邮件，这时会无缝登陆到邮件服务器发送邮件给目的地，请记住，邮件服务器是一台或者是一组多台服务器组合起来的，搜索服务器同样也是一台或者一组多台服务器组合起来的，它们之间靠网络连接，它们彼此之间有域名，有 IP 地址有端口号，彼此之约定好固定的数据格式就可以了。

对单独的服务器就是 PC，有独立的 CPU，独立的以太网，有独立的硬盘以及内存，单台服务器符合冯诺依曼体系结构，在这个单独服务器上，国内目前通常部署的是开源 linux 操作系统，在这个上面部署了邮件系统，或者搜索系统，而这些系统是独立可以运行的，监听服务器的端口，有自己的 IP 地址和域名；这些系统可以被称为一个独立的进程或者独立的软件项目，有一个单独的 main()函数，多名研发人员组合在一起服务开发这个软件项目，这个软件项目就是第一种类似 windows 的大项目，里面会按功能分组或者按照软件层次分组，具体如何分组会根据产品的功能不同，特点不同而具体划分，例如百度搜索里会有专门的算法专家，让用户输入的关键词最优匹配到后台巨大的数据库，按照优先级把用户想要的数据准确呈现在用户的面前，如果算法不好，会让用户搜索一次过多访问服务器使得搜索时间变长用户体验不好，也会让用户想搜索美食时，弹出旅游的内容，用户如果搜索不到想要的资料就会流失掉。可以看到，大项目内部不要只觉得大，其实内部分工非常细，每个细分工作都由多名研发人员在背后默默努力的贡献自己的智慧。

6.5 以人为本，从实际出发

软件领域和硬件领域的发展都是非常有前途的，两者混合都懂的在目前国内实在为数不多，懂硬件的不怎么懂软件，懂软件的一般都很少懂硬件；在学习的过程中，心态要摆好，不要贪大贪全，什么都想学，学习需要一步一步一个脚印一个脚印的走过来。

所有的高手，都是这样的，首先要从自己个人的优势出发，比如有的人是学软件出身的，有软件的背景，那么软件就是你的强项，你不断加强自己的强项会带来许多的成就感，把软件达到一定境界之后，有了个饭碗或者真正的上手之后，有闲暇的时间，就可以抽取出来一部分来研究和学习硬件。硬件背景的同学也是同样的道理，一定要从实际出发，逐步深入。

再深入说一点，拿我们的神舟 51+ARM 单片机开发板来说，你软件比较有优势，以前在学校就是学这个的，那么你通过这些例程的学习，会对硬件有一个初步的认识，玩通软件的过程，也就是对硬件的一种认识，把 51 单片机的所有例程都在一遍之后，如果软件的代码都掌握了，知道怎么阅读芯片手册，数据手册查看寄存器了，那么其实就是往硬件的方向靠近一步了。当你做得做了，硬件就会越来越了解，而在这个过程中，你又学会了更多的软件知识，因为你的软件背景，使得你掌握软件的速度要比一般人快很多，这样建立起来的成就感会让你越战越勇。反之，如果软件背景的一上来就使劲研究硬件，研究芯片手册，研究 PCB 布线，认为首先要弥补自己最不擅长的方面，那么就会陷入效率不高，挫败感，并且还学不到多少东西的境地，这个学习方法和心态希望大家细细去体会。

这是学习的门道和内功心法，这是学习的九阳神功，有我们这本好的书，一定要配这样一套好的心法辅助才能蒸蒸日上！本书因时间仓促，还有水平有限，有许多不足和欠缺之处，希望大家多多指正。